CSProX 4.1

(Last Update: March 22, 2012)

NEW FEATURES

GPS Function for WIN32

This function provides information from the attached GPS (latitude, longitude and altitude). The function is able to search a valid GPS. See a full description below.

Interview Recording and Playing (WIN32 only) WARNING: This feature does not work with some specific sound cards.

Format:

CSEntry AppName.ent [**/UseVoicePlayer**] [**/UseVoiceRecorder**]

or

CSEntry AppName.pff [**/UseVoicePlayer**] [**/UseVoiceRecorder**]

When **/UseVoiceRecorder** is used, the system will record the interview (ogg format) using the default computer input device (it should be the microphone). The ogg file name has the following structure:

DatFileName.**KeyValue**.ogg

where **DatFilename** is the name of the data file and **KeyValue** is the corresponding case key.

During the recording process the file with extension “ogg.track” is generated. That file keeps information about the position where the questions start in the corresponding “.ogg” file.

When **/UseVoicePlayer** is used, the system will use the corresponding “.ogg” and “ogg.track” files to play the recorded interview. This last file is used by the system to seek to the position where the corresponding question starts in the “.ogg” file. /**UseVoicePlayer** is used in **modification mode** by the supervisor to hear the interview.

By default the system will record only 600 seconds (10 minutes) by question. In case you want to change the default settings you can use the function VoiceRecorder (see below).

VOICERECORDER Function (WIN32 Only)

Description:

This function is used to control the recording process. Most of the times it's not necessary to use this function because the recorder will start automatically when the **"/UseVoiceRecorder"** parameter is used in CSEntry.

Format:

n = **VoiceRecorder (Start | Pause | Status |TimeOut, TimeOutValue);**

| indicates that one or the other keyword may be used

- **N = VoiceRecorder (Start);**

The function returns 1 if the recorder can be re-started (continue) after a pause or 0 when it fails.

- **N = VoiceRecorder (Pause);**

The function returns 1 if the recorder can be paused or 0 if it failsl.

- **N = VoiceRecorder (Status);**

The function returns the current recorder status: 1 if the recorder is working (running) or 2 if the recorder is stopped (paused) or it's not working.

- **N = VoiceRecorder (TimeOut, TimeOutValue);**

The function changes the maximum recorder time (600 seconds by default by question). TimeOutValue is the new timeout in seconds. When the TimeOutValue is reached the system will pause the recorder until the user moves the cursor to other question. The function returns the TimeOutValue.

Upload/Download Files tools

CSProX has 2 programs to realize the transference of files: **CSFileSyncClient.exe** and **CSFileSyncServer.exe.**

CSFileSyncClient corresponds to the client of transmission and it is used to send/receive information to/from a server. The servers to which it is possible to connect are the server CSFileSyncServer, an FTP server or a Pocket PC connected via ActiveSync.

CSFileSyncClient and CSFileSyncServer run under WIN32 and Mobile Windows (5/6). This way it is possible to transmit the data files directly from a Pocket PC on to a remote FTP server via internet or also transmitting the data from a Pocket PC to another. The remote FTP server may be running in Linux and/or Windows.

Additionally the CSFileSyncClient for WIN32 is able to communicate directly with ActiveSync with no need to have a server in the Pocket PC (Ftp Server or CSFileSyncServer).

CSFileSyncClient uses the parameter file **CSFileSyncClient.ini** (located in the same folder as the executable) to determine the actions it will follow.

CSFileSyncServer uses the parameters file **CSFileSyncServer.ini** (located in the same folder as the executable). Additionally the parameter **"/Start"** can be used for the program to start immediately without showing the initial dialog of confirmation.

Using these programs it is possible to recover information from remote computers and to refresh the applications.

See in www.serpro.com (download section) a complete description of those utilities.

MAP Function

Description:

The map is a kind of array that is accessed by an alphanumeric key. The MAP function is used to manipulate the maps.

Each map has a name and it doesn't need to be declared. The map name needs to be passed as parameter to the MAP function. More than one map can be used per application.

Format:

Ret = MAP(GET, AlphaExprMapName**,** AlphaExprMapKey);

Returns the value associated to the map key specified in *AlphaExprMapKey.* Returns NOTAPPL if the key is not found in the map.

Format:

Ret = MAP(SET, AlphaExprMapName**,** AlphaExprMapKey, NumExprMapValue);

Assigns the value specified in *NumExprMapValue* to the map key specified in *AlphaExprMapKey.* The function returns 1 if the value is assigned or 0 if it's fail.

Format:

Ret = MAP(DEL, AlphaExprMapName**,** AlphaExprMapKey);

Deletes the map key specified in *AlphaExprMapKey.* The function returns 1 if the key was removed from the map 0 if it's fail.

Format:

Ret = MAP(INIT, AlphaExprMapName**,** NumExpresionMapInitialSize);

Clean up the map and set up the initial map size (*NumExpresionMapInitialSize*).The function returns 1 if the map was reinitialized or 0 if it's fail.

Format:

Ret = MAP(LIST, AlphaExprMapName**,** AlphaNumericArray);

Copy (and sort by value) the values from the map to an alphanumeric array. The function returns the number of elements copied or 0 if there are no elements or fail.

AlphaExprMapName is an alpha expression containing the map name.

AlphaExprMapKey is an alpha expression containing the map key.

NumExprMapValue is a numeric expression containing the value associated to the corresponding map key.

NumExpresionMapInitialSize is a numeric expression containing the initial map size. This function could be useful when the map contains a large amount of elements (it's more efficient to set up the initial estimated size from the beginning to minimize the internal map memory reallocations). This value doesn't represent the maximum amount of elements to be handled by the map. Theoretically, the map doesn't have a maximum amount of elements.

AlphaNumericArray is a one dimension alpha numeric array declared in GLOBAL procedure

Example: Detecting and removing duplicated cases from a file.

In the example below we use the map "MyMap" to keep all the keys from the data file. Using the function KEY(MYDICT), we get the current case key from the input data file associated to

the dictionary MYDICT.

```
PROC HOUSEHOLD
PreProc
 {Get the map value associated with the current key}
if Map( GET, "MyMap", KEY(MYDICT) )=1 then
  errmsg( "Case Duplicated");
  skip case;
else
 {Set the map key associated with the current case key with the value 1}
  Map( SET, "MyMap", KEY(MYDICT), 1 );
endif;
```

AMAP Function

Description:

This function is similar to MAP function but works with alphanumeric elements. The function returns an alpha string. It's only used with the **GET** parameter.

Format:

alphaRet = AMAP(GET, AlphaExprMapName**,** AlphaExprMapKey);

Returns a string containing the value associated to the map key specified in *AlphaExprMapKey.* Returns an empty string if the key is not found in the map.

To assign some value to the map, it's necessary to use the **MAP** function with the following format:

Ret = MAP(SET, AlphaExprMapName**,** AlphaExprMapKey, **AlphaExprMapValue**);

Assigns the value specified in ***AlphaExprMapValue*** to the map key specified in *AlphaExprMapKey.* The function returns 1 if the value is assigned or 0 if it's fail.

To delete or initialize the alphanumeric map, it's necessary to use the **MAP** function with the parameters **DEL** or **INIT** respectively.

Comments in the logic

Description:

Now it's possible to use "//" in the logic to comment until the end of the line.

Example:

```
PROC SEX
  // This is a comment

 if $ = 1 then // This is a comment
  {You can also use the traditional comments}
endif;
```

TOUPPER and TOLOWER functions

Description:

Those functions are used to transform an alpha expression to uppercase or lowercase respectively.

Format:

alphaRet = tolower | toupper(alphaExpresion);

Converts to lower (upper) case the string specified in **alphaExpresion**.

Example:

```
Errmsg( "%s", tolower ( "JoHn" ) ); // Will display "john"

Errmsg( "%s", toupper ( "JoHn" ) ); // Will display "JOHN"
```

GETMACHINENAME Function

Format:

alphaRet = GetMachineName();

The function returns the machine name.

Example:

```
If GetMachineName() = “SUPERVISOR” then
        // Do Something
Else
        //Do Nothing
Endif;
```

OTHER FEATURES

ABS Function

Format:

Ret = ABS(Arithmetic-Expression);

The function returns the absolute value of an arithmetic expression.

Example:

```
X = ABS( A – B );
```

X is equal to the absolute value of the difference between A and B.

Alpha Parameters in Functions(*)

User’s Functions accept alpha parameters defined in functions declarations as follows:

```
function Function_Name( alpha(32) Var_Name, Var_Code )
…
end;
```

Alpha User's Functions (*)

Alpha user’s function can be defined. This means that the function will return a string of characters instead of a numeric value as a normal/numeric function.

The format of an alpha function is as follows:

Function alpha(L) function_name(par1, par2, …, parn)

Where:

L: is the length of the function or the length of the character string that the function returns;

Par1, par2, …, parn are the various function parameters.

ASSERT Function

This super function combines the functionality of two different functions to facilitate the consistency checking, requiring almost no programming skills. The function is as follows:

ASSERT(Logical-Expression)

[**OnError**([**WARNING**|**ERROR**] [,MsgNum] [,MsgText] [,**CONTINUE**])]

[**INCLUDE**(ItemListLiteral)];

Where:

Logical-Expression → Expression Relational-Operator Expression
[Logical-Operator Expression]

Expression → Numeric-Expression | Alpha-Expression | Logical-Expression

Numeric-Expression → Numeric-Symbol [Arithmetic-Operator Numeric-Symbol]

Numeric Symbol → Numeric-Variable | Numeric-Constant | Numeric-Function

Arithmetic-Operator → +- plus and minus operators
*/ multiplication and division operators
% integer part of the module division complement
^ power operator
() group operators to break the default priority of the arithmetic operators and/or the logical operators

Logical-Operator → and &
Or |
Not !
⇨ Implication operator (=>)
⇔ If and only if operator (<=>)

The logical expression is evaluated and in the event the evaluation is FALSE, the OnError

clause is executed (if exists).

If the OnError clause is not specified, the system generates an error message where all the variables involved in the logical expression are displayed. Both the long name (label) and short names are used to identify the variables and the current contents/values displayed.

Special attention deserves the *implication operator* (=>) and the "*if and only if" operator* (⇔) since they contribute to make the function even more powerful. Take for instance the following ASSERT function:

Assert(A => B)

Where 'A' is a logical expression as complex as needed and 'B' a similar expression; if the expression 'A' is TRUE, then the expression 'B' is evaluated; if 'B' is FALSE then the OnError clause would be executed

OnError:

- The WARNING or ERROR keywords can be used as title of the error message (as shown below in the example); the default keyword is ERROR
- The message number is a numeric constant identifying an error message in the error messages file.
- Message Text is an alphanumeric expression displayed as error message; this option overwrites any text in the error message file having the same message number.
- **Continue** is a keyword used to provide the interviewer the option to leave the conflict or consistency error without fixing it and continue with the next question/field. This option will be clear after the remaining part of the function is explained. However, the user should be aware that in the event that the Continue keyword is not specified, the interviewer will be forced to fix the conflict between the different variables in the logical expression.

Include:

- The include clause is used to specify either variables that are not part of the logical expression or to further enhance the name/description of the variables when needed. The Include clause should specify the name of the variable(s) followed by the label we want to display instead of the data dictionary label.

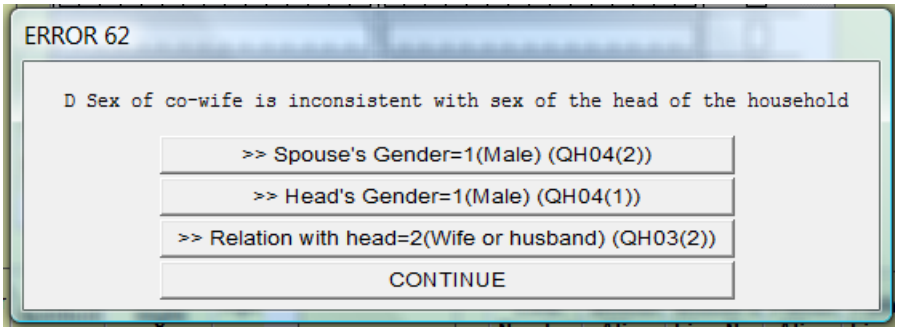
As stated before, if the logical expression is evaluated to FALSE, the OnError clause –if present- will be executed and, in any event, all the variables involved in the logical expression are displayed. The example shown below illustrates the use of the function and subsequently, the format the error message is displayed. To make the data consistent, at least one variable of the logical expression needs to be fixed. In our example shown below, the ASSERT function is checking that "if the relationship to the head of the household of the current occurrence of the household members roster is ***spouse*** the gender of the current occurrence has to be different from the gender of the head of the household". If both genders are the same, there is an inconsistency that needs to be fixed by changing one of the three variables: (i) sex of the spouse (QH04(i)), sex of the head of the household (QH04(1)) or the relationship to the head of the household (QH03(i)) to something different from ***spouse***.

Another merit of the ASSERT function is that by clicking on the button where a variable is displayed, the system automatically position the cursor on the corresponding field. Thus, no navigation is required to go back to the desired variable. Returning to the current position is just one click away.

The following is an example of the assert function:

```
assert(QH03 = 2 => QH04 <> QH04(1))
        onerror ( 62, Continue )
        include( QH03, "Relation with head", QH04, "Spouse's Gender",
                QH04(1), "Head's Gender" );
```

The error message dialog would be as follows:



ATTRIB Function

Format:

n = **attrib**(Symbol_Name, Key-Word);

This function provides information about a specific object of the Data Dictionary or the Form File –***Symbol_Name***-. The type of information requested –**Key-Word**- depends on the object specified under **Symbol_Name**. If the keyword specified is not applicable for the specified symbol name, the function returns '-1'.

The following objects can be specified under **Symbol_Name**:

- Data Dictionary Name (DDN)
- Record Name (RN)
- Item/Sub-Item Name (IN)
- Field Name (FN)
- Group/Roster Name (GN)
- Table Name (TN)
- Application Name (AN)
- Any Object (AO)

Symbol-Name	Key-Word	Function Returns
AO	PARENTSYMBOL	Parent's Symbol Number in Symbol Table
AO	SYMTYPE	Symbol Type Code: See symbol types at the bottom of this table
AO	SYMBOL	Symbol Number in Symbol Table
RN or IN	LEN	Record, item/sub-item length
IN	DEC	Number of decimals
IN	FMT	1: Numeric; 2: Alpha (upper); 3: Alpha
IN	START	Item starting position
IN or GN	MAXOCC	Max number of occurrences of object
IN	NUMVALUESETS	Number of value sets of the item
IN	HASDECCHAR	Number of decimals; -1 if not applicable (object isn't item)
IN	ISITEM	0: Object is not an tem; 1: Object is item; -1: Not Applicable
IN	ISSUBITEM	0: Is not a sub-item; 1: Is a sub-item; -1: Not Applicable
IN	ISCOMMON	0: Item isn't part of Common; 1: Is part of Common
IN, GN, TN	NUMDIM	Number of dimensions of object
FN	ISPROTECTED	0: The field is not protected; 1: The field is protected

FN	ISSEQUENTIAL	0: The field is not sequential; 1: The field is sequential
FN	ISSTICKY	0: The field is not sticky; 1: The field is sticky
FN	ISUPPERCASE	0: The field is not uppercase; 1: The field is alpha-uppercase
RN	ISREQUIRED	0: The record is not required; 1: The record is required
RN or DDN	NUMITEMS	The number of items of the record or Data Dictionary
DDN	NUMLEVELS	The number of levels of the Data Dictionary
DDN	RECTYPESTART	The starting position of the Record Type
DDN	RECTYPELEN	The Record Type length
DDN	NUMRECORDS	The number of records of the Data Dictionary
GN or TN	NUMROWS	The number of rows of object
GN or TN	NUMCOLS	The number of columns of object
GN or TN	NUMLAYERS	The number of layers of the object
GN	ORIENTATION	1: Horizontal; 2: Vertical; -1 Not Applicable
DDN	DATE	The DDN creation date (YYYYMMDD)
DDN	TIME	The DDN creation time (HHMMSS)
AN	NUMDICTS	The number of DD of the application

Symbol Type Codes:

Dictionary	0
Section	12
Variable	14
Work-Variable	16
Form	17
Record	26
Group	27
Value Label	28
Value Set	30
Relation	31

AUTOCODE Function

n = **Autocode**(*Alpha_Var, Separator-List, Dif-Number, Inverted-File-Name, Codes-Description, Exclusion-List[, History-File-Name] SET array-name);*

[ ] indicates that this part is optional.

The function returns the following codes:

>0: The code description specified by Alpha-Var renders only one code and is automatically returned by the function.

-1: The code description entered didn't match any of the original text list specified in the "Code-Description" parameter (the description is too vague).

-2: When the number of alternatives found by the algorithm is more than one but less or equal the maximum specified by the SET AUTOCODE command, the dialog box displaying each of the alternatives is open; if the operator is not satisfied with any of the alternatives can press the <Esc> key and the function will return -2.

-3: The number of alternatives rendered by the matching process is higher than the maximum specified by the SET AUTOCODE command. In this case, there is no dialog box displaying the various alternatives and the only choice left to the operator is to further refine the code description entered.

This function, also call "Assisted Coding", hides all the complexity of the automatic coding process, allowing the user to pass the text to be coded, and receiving from the function the corresponding code, provided that it is a valid description of the field to be coded.

The parameters list passed to the function is as follows:

- **Alpha_Var** is the text or description of the item to be coded; it can be in capital letters or upper and lower case; it can contain any type of ASCII characters but before it is submitted to the matching process, it will be **normalized**. By normalization we should understand the following operations: (i) conversion from lower to upper case; (ii) special characters like 'á', 'é', 'í', etc are converted to 'A', 'E', 'I', etc.; (iii) special characters passed in the **Separator-List** –second parameter- are ignored; (iv) excludes from the text all words that are found in the **Exclusion-List** –sixth parameter-; (v) the remaining words are sorted in ascending order creating a new list of independent words that will be submitted to the matching process; the first matching process step will be carried out over the **Historic-file** (if exists) –seventh parameter- to check if that description already exist.
- **Separator-List** is a list of special symbols (character string) that should be considered as words separators but otherwise, they should be ignored at the time the text is scanned. This list will probably include symbols like ',', '-', '(', '}', etc. This parameter is more adequate when the function is used in batch rather than in data entry since the DE operators and interviewers can be instructed not to use these special symbols when entering the text.
- **Dif-Number** is a numeric constant/variable that tells the systems the number of different characters that are acceptable -in the word matching process- in the event that a given word is not found in the data base (inverted file). Normally, this constant is one or at the most two, allowing for misspellings, plural/singular, masculine/feminine, etc. The algorithm used is the Levenhstein algorithm having the advantage of being independent of the language. Essentially, given a word (keyword), a list of words or dictionary (the inverted file), and a constant N, it will give a list of all the words in the dictionary that have at the most N differences (in characters) with the word being searched (keyword). It's clear that using 2 differences might lead to wrong conclusions and we highly recommend you test thoroughly the algorithm before applying it to the real process.
- **Codes-Description** is the original ASCII file where each possible code is fully described. The file is used first to produce the inverted file and, later on, to display different alternatives when the text entered to auto-coding renders more than 1 option. In these cases, all valid alternatives are displayed for operator's selection.
- **Inverted-File-Name** is the name of a special file that has to be generated prior to the use of the AUTOCODING function. The file is produced by a stand alone utility described later in this documentation. This file and its organization is the essence of the automatic coding strategy and algorithms.
- **Exclusion-List** is a list of words (alpha array) to be excluded in the text normalization procedure and it should be identical to the one used in the stand alone utility to generate the inverted file. Usually, this list includes words like articles, prepositions, conjunctions, and

generally speaking, words that have importance from the grammar point of view but don't add meaning to the text.

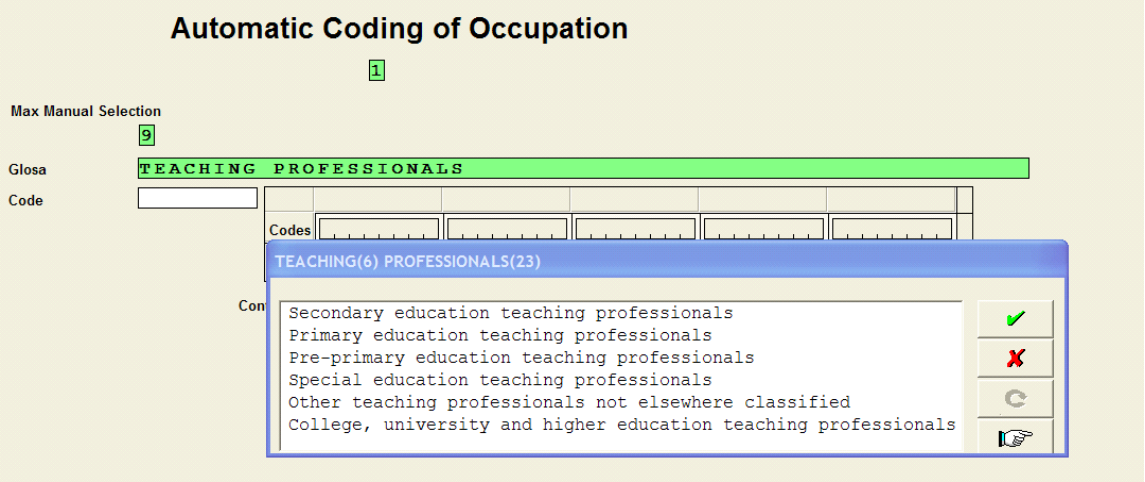
- **History-File-Name** is the name of the file were the normalized text and the corresponding code either automatically imputed or assigned by the interviewer or DE operator will be stored. Conversely, before starting the matching process, the normalized text is used as index to this file and if the key is found, the corresponding code is assigned to the current text.
- **SET array-name** tells the system the array name where the codes of the different alternatives will be stored when more than one option was rendered by the matching process.
- In addition to the above mentioned parameters, this function works in conjunction with the SET AUTOCODE command to define the maximum number of possible alternatives the user wants to display. If the description entered renders more than the maximum specified by the SET command, the function will return a -3 code, meaning that the description is vague and therefore needs to be revised.

Operation Description:

As it was mentioned before, the text to be coded is normalized and converted into a list of independent and sorted keywords (in ascending order) that typically will produce three to four words (there is no limit to the maximum number of keywords). The first step is to check if the concatenation of keywords used as a key already exists in the historical file. If it's found, the corresponding code is returned by the function. Note that the historical file is optional and is created based on similar studies (surveys, censuses) previously carried out. Given the role importance this file plays, it is crucial to make sure that it's free of errors or they will contaminate the current operation. Another consideration regarding the historical file is that the code description should be sufficient to determine a unique code without the need to use one or more dependent variables in the process.

The second step consist in searching each keyword in the inverted file, keeping track of (i) the total number of hits or different original codes where that word was found; (ii) the list of actual codes where the word was found. If a keyword is not found in the inverted list, it will be omitted from the third step explained below, and there will be no automatic imputation even in the event when the other keywords all have in common one code.

The third step consists in the determination of the common code(s) of all the lists generated in the second step (intersection of code sets). Three different situations can be faced when the final code set is generated: (i) the code set is an empty list, in which case, the system will require assistance from the DE operator to further refine the description entered or in selecting the most adequate option; the possible options will be taken from the union of the code sets of each keyword, displayed by number of matches –those codes that had more words matching will be displayed first-, with the matching words highlighted; (ii) the code set has only one code, in which case the function automatically returns that code; (iii) the intersection list has more than one code, in which case, again the system will require the operator's assistance to identify the most likely one from the different alternatives. In this case –as in (i) above-, the options will be displayed in a box having as heading the original gloss entered by the operator.

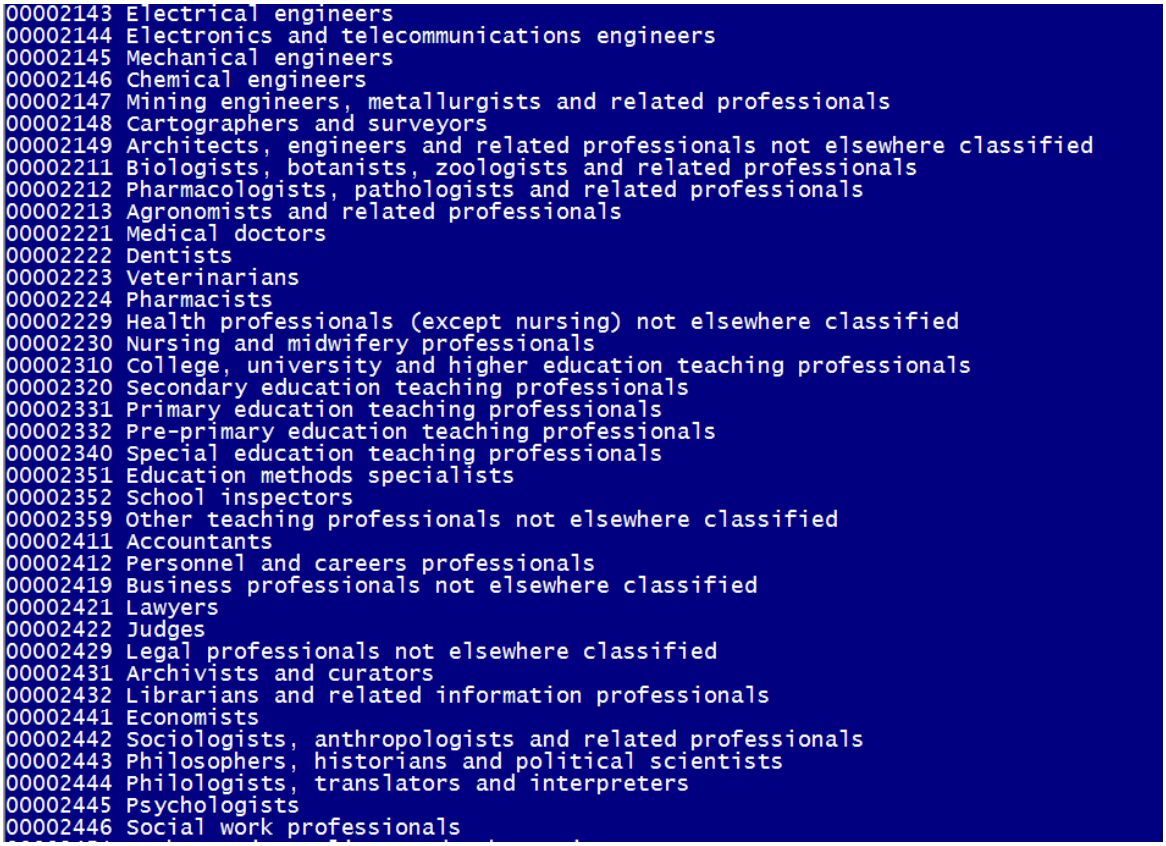


In the example above, the occupation description entered "TEACHING PROFESSIONALS" is clearly vague since there are six equally possible cases that match the specification. It's important to point out that the occupation description used in this example is not suitable for an automatic coding process since the wording used in the description is more academic than the every day wording. Probably, the normal answer for the specific occupation would be "HIGHSCHOOL TEACHER" or "UNIVERSITY PROFESSOR" rather than teaching professional. The results that will be obtained in the automatic classification depend strongly on how suitable the original list is for this process. In the particular case of occupation, it is highly advisable to have a detailed list of all occupations even if the same code has to be repeated several times with different descriptions. It is clear that a list description aimed to do manual coding is not suitable for automatic coding. Let's consider the following descriptions: "OTHER PROFESSIONALS NOT ELSEWHERE CLASSIFIED". This description might mean something for a person reading it in a specific context but in our case, it doesn't mean anything.

Whenever the dialog box is displayed, the Interviewer/coder can either select one option, in which case the function will return the corresponding code, or press the <Esc> key refusing to select one, in which case, the function will return the code -2.

In the rare event that even the union of all the individual keyword codes lists is empty, the function will return the code '-1'.

One crucial file for all this process is the original "Code_Description" since from there the inverted file is generated. The following image shows part of this important file:



Although as it has been anticipated, the code description presented here is not adequate for an automatic coding process, it shows some characteristics that have to be pointed out.

- The code (first column shown in the figure above), has to be numeric and of fixed length; if there are codes of different length, they should be padded with zero(s) on the left. This is important since at the time the file is scanned to produce the inverted file, the code length is defined. The code length is calculated using the first line only. Thus the file needs to be homogeneous (all codes have the same length). The first BLANK or space following the code is used as delimiter denoting the end of the code string.
- The text following the numeric code string is the code description and should be as concise and thorough as possible. However, this text is the one that will be displayed when more than one alternative is rendered by the matching process. Therefore, it should be clear enough for the operator to decide between the various alternatives.
- The file should be an ASCII file.

Boolean Key Words

The Boolean key words TRUE and FALSE have been added to the CSProX language. The **TRUE** value is equivalent to 1 and the **FALSE** value is equivalent to 0.

Example:

```
If  A = TRUE and B = FALSE then
  skip to Q320;
endif;
```

The above command sequence is equivalent to:

```
If A = 1 and B  = 0 then
   skip to Q320;
endif;
```

CHECKDATE Function

Format:

d = CheckDate(DD, MM, YYYY);

where DD is the day, MM the month and YYYY the year of the date that want to be checked.

Description:

The CheckDate function verifies that the date passed (DD, MM, YYYY) is valid; that means that the day is consistent with the month and year. It returns 0 if the date is invalid and the day number based on a remote date (year 0 or after)). Besides checking the date, the function facilitates the calculation of date intervals in days.

COUNT Function Extension

Format:

N = **count**(Group-Name [**where** Logical-Expression] [**Set** Numeric-Array]);

[ ] indicates that this part is optional.

A numeric array can be also passed as parameter to be set with all the GROUP occurrences that satisfy the **WHERE** condition. Therefore, N is also the number of occurrences of the array that are set by the **COUNT** function showing the individual occurrences meeting the **WHERE** condition.

The array is **zero based** meaning that the first occurrence meeting the WHERE condition is found in occurrence '0' of the array, the second in occurrence 1 and so on.

Example:

```
n = count( HH_Members_Roster where( Sex = Female and Age > 15 ) set
      Females_Array );
```

Once the function is executed, ‘n’ will have the number of females older than 15 years and the array "Females_Array" will have the occurrences/lines number of each of them.

ENTER Command Extension

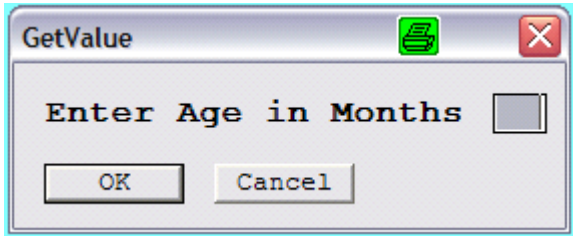
As opposed to the previous command format, where only external dictionaries with form(s) (Form File Name) could be specified, the extension allows to capture the value of one particular item on the fly.

ENTER Item-Name [, Title];

Item-Name can be a dictionary item or an application declared variable. Title is optional and allows specifying a text in the data capture dialog box.

Example:

ENTER Age, "Enter Age in Months";



The content of the edit box is initialized with the current item value.

Note: The use of this command is oriented to interactive applications only (CAPI or DE). When running the ENTRY in BATCH, the ENTER command is not executed and thus, the execution flow might differ from the entry depending on the purpose of the data capture.

Errmsg Function Extension*

Two extensions have been added to the **ERRMSG** function: the first one is that the correspondence between the parameters specified in the ERRMSG function and the parameters specified in the message file is checked automatically by the compiler at the time

the application is executed; the second, the **SELECT** clause described below.

Format 1:

```
  [b =] errmsg(string-exp[,p1[,p2[,...,pn]]])[denom=var]
[case|summary]
              [select(Alpha-Var1[, Alpha-Var2, Var_Name2, … , Alpha-
          Varn, Var_Namen | Continue])];
```

[ ] indicates that this part is optional.

| indicates that one or the other keyword may be used.

Format 2:

```
  [b =] errmsg(msg-num[,p1[,p2[,...,pn]]]) [denom=var] [case|summary]
              [select(Alpha-Var1, Var_Name1[, Alpha-Var2, Var_Name2 …
          , Alpha-Varn], Var_Namen | Continue)];
```

[ ] indicates that this part is optional.

| indicates that one or the other keyword may be used.

msg-num can be a number or numeric expression

Description:

The errmsg function displays a message on the data entry screen (when used in a Data Entry application) or writes a message to the batch edit report (when used in a Batch Edit application). If messages are defined via the message number ("msg-num"), then those messages will be stored in a message file [.mgf].

Each parameter (e.g., "p1") is sequentially inserted into the error message. Parameters can be numeric or alphanumeric expressions, but the type of parameter must match the type of the corresponding format code type in the message text. The maximum number of parameters in an errmsg function is 20.

In the message text the format codes can be:

%[*n*]d	=	Insert a number and display it as an integer
%[*n.d*]f	=	Insert a number and display it as a decimal value
%[*n*]s	=	Insert a text string

"n" is the number of characters inserted and "***d"*** is the number of decimal places to show for a number.

Numbers are never truncated.

If "***n"*** is positive, the insert is right justified in the size of the field. If "***n"*** is negative, the insert is left justified in the size of the field. If "***n"*** is a positive number with a leading zero, the insert is right justified in the size of the field and zero filled to the left.

When inserting a number, if "***n"*** is preceded by a **+**, the sign of the number is always displayed.

The **select** clause should be used in DE applications and it allows displaying, together with the error message, a set of buttons related to different actions the DE operator can select. The number of buttons is equivalent to the number of *Alpha-Vars* specified in the **select** clause and the text shown in each one of them is the corresponding content of the *Alpha-Vars.* The Var_Name1,…,Var_Namen is the input variable name PROC to which we want to go when the corresponding button is clicked/selected by the operator. Note that this will normally be equivalent to an implicit "REENTER" command to Var_Namei. The value "n" returned by the errmsg function is equivalent to the button number selected by the DE operator. The following figure illustrates the use of this clause:

Examples:

Value = 23456	%d	`23456`
	%10d	`  23456`
	%-10d	`23456`
	%010d	`0000023456`
	%+10d	`  +23456`
	%+010d	`+000023456`
	%f	`23456.000000`
Value = 12.567	%f	`12.567`
	%10.3f	`  12.567`
	%-10.3f	`12.567`
	%10.2f	`  12.57`
	%10.5f	`12.56700`
	%010.3f	`000012.567`
	%+10.3f	`+12.567`
	%+010.3f	`+00012.567`
	%d	`12`
Value = "abcdef"	%s	`abcdef`
	%10s	`  abcdef`
	%-10s	`abcdef`

The **denom** keyword allows you to specify a denominator, so that you can show percentages in the summary portion of the output listing. This is very useful for showing edit failure rates. In Example 2 below, the output listing will show the number of times there was more than one head of household divided by the number of households processed during the run. Note that it is the responsibility of the application designer to write logic to put the proper values into the denominator variable.

The **case** and **summary** keywords give you some control over the output listing. By default, the output listing shows you messages case by case, and also shows you a summary of the number of times the message was triggered (with an optional denominator, described above). You can limit the output listing to only case-by-case reporting, or only summary reporting by using these keywords.

Return Value:

The function returns a logical value 1 (true) if successful and 0 (false) otherwise. If the **select** clause is used, it returns the button sequence number

Format 1 Examples:

Example 1:

```
errmsg("Head of household is %d years old.", AGE);
```

Example 2:

```
errmsg("More than 1 head of household") denom = PERSON_COUNT summary;
```

Format 2 Example:

```
OK = errmsg (1,"June",30,31);
```

where the message file contains the following text:

```
1 %s has only %d days. You entered %d!
```

Note the errmsg call could have also been invoked as follows:

```
i = 1;
OK = errmsg (i,"June",30,31);
```

Select clause Example:

```
n = errmsg("Conflict between age of mother (%2d) and age of child (%d)", Age_Mother,
          Age_Child) select( "Fix Mother's Age", Age_Mother, "Fix Child's Age",
          Age_Child, "Ignore the Error", Continue);
```

An error message with three displayed options is issued; the three options are: to fix the mother's age, to fix the child's age or to ignore the error. Depending on the button clicked by the interviewer/operator is the option executed. The first two are options to direct the flow to the "Mother's age" procedure or to the "child's age procedure" and the third will simply ignore the error and continue with next field procedure/s.

EVAL Function

Format:

n = **eval**(**Key-Word**, Code-list[, Code-list1, Code-list2]);

Depending on the keyword used, this function performs the following arrays operations between **Code-list1** and **Code-list2**, storing the result in the array **Code-list**: (i) intersection of both arrays; (ii) "exclusive or" of both arrays; (iii) union of both arrays and (iv) sort of one array.

Code-list, Code-list1 and Code-list2 must be declared arrays.

The following is a list of the different alternatives of the EVAL function:

- **Intersection**

  n = **eval**(**intersection**, A0, A1, A2);

  The intersection between A1 and A2 is placed in A0.

- **Union**

  n = **eval**(**union**, A0, A1, A2);

  The union of both arrays A1 and A2 is placed in A0.

- **Exclusive or**

  n = **eval**(**minus**, A0, A1, A2);

  All elements that don't belong to both arrays A1 and A2 are passed to A0.

- **Sort | Sortd**

  n = **eval**(**sort**, A0);

  The function will sort –in ascending order- the list of elements stored in array A0. The same array is used to return the sorted list.

  n = **eval**(**sortd**, A0);

  The function will sort the list A0 in descending order.

Field Properties Extension

As it can be observed in the dialog box below, the "Auto-complete" field property has been added to the normal CSPro field property dialog box. The purpose of this new property is to add a new class of alphanumeric field oriented to facilitate the lookup table and automatic coding of questions like "Geographic Location" (State, Province, etc.) or port names, institution names, etc. without the need of a programmatic effort from the application developer.

Concentrating on the left side of the auto-complete dialog box, we can see three radio buttons (mutually exclusive options) as follows:

No: Meaning that the auto-complete property will not be applied to the field (this is the default option).

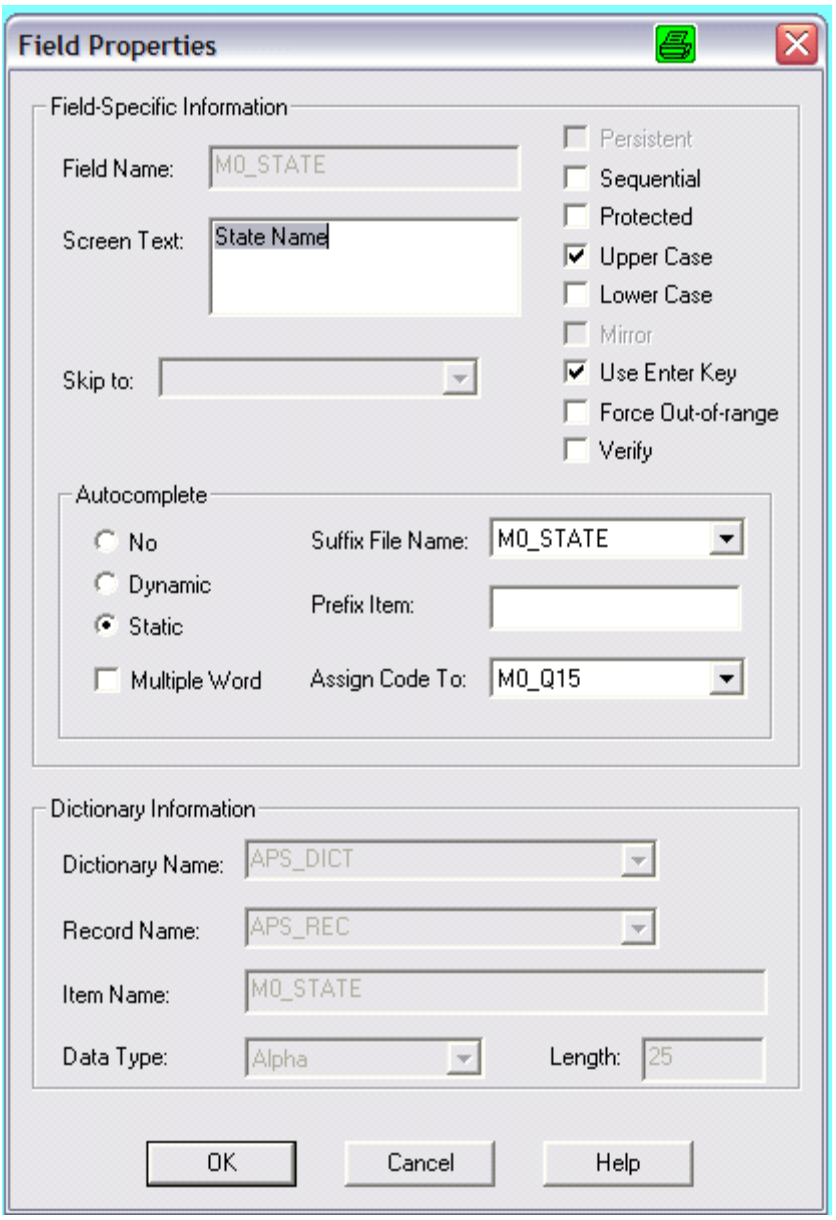
Dynamic: Meaning that the auto-complete property is applied to the field and in addition, the lookup list/table is dynamic. Dynamic means that if the name doesn't exist in the lookup table it will be added to the list.

Static: Meaning that the name entered by the operator/interviewer should exist in the lookup table; if it doesn't exist, the name is wrong and should be reentered.

In addition to the three radio buttons, a check box can be checked indicating that the field has or might have **multiple words** and that each one of them should be in the lookup table. This property has been added with the automatic coding of variables (i.e. Occupation) in mind. Thus, it should only be checked in conjunction with the "Static" attribute.

Looking at the right side of the auto-complete dialog box we find all the information we need to provide to fully identify the lookup table and its structure. Lookup tables are very simple data files containing a text/name and optionally, a numeric or alphanumeric code.

The lookup table file name has the following structure: "<Data-Dictionary-Name>.<Field-Name>.HST". Since different data dictionaries might have common field/item names, it's necessary to prefix the DD name before the field name. However, if only the field name is provided, the current DD name is assumed and prefixed by the system. In addition, if the lookup table file name coincides with the current item name, it can be left blank and both the DD name and the item's name are assumed. Note that two or more different fields might use a common lookup table (e.g. State of birth and "State of Residence"). Thus, we are free to link the "State" lookup table to either field.

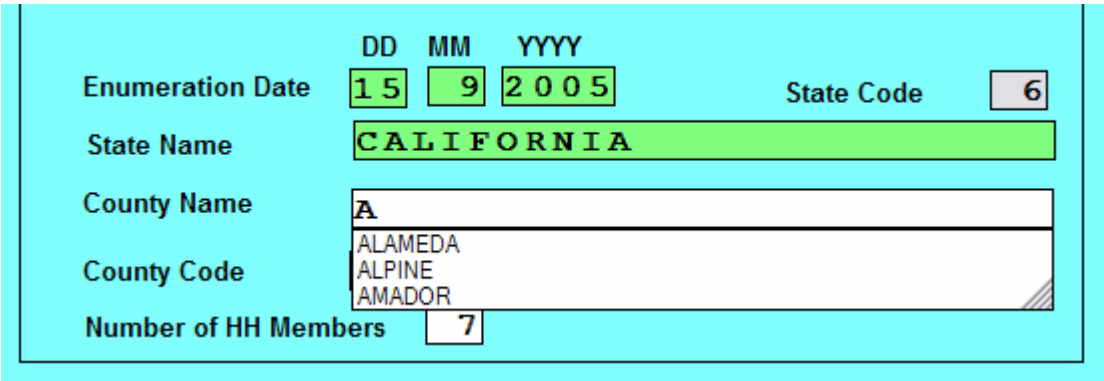


The following information might be filled in according to the needs:

Suffix File Name: This is the name of one of the DD alphanumeric items that is linked to the lookup table. If not provided, the field name is assumed. As mentioned before, the item name specified is part of the lookup table file name associated with the current field name.

Prefix Item: This is the name of an alphanumeric working variable containing a prefix that combined with the content of the field (e.g. M0_STATE in our example above) will be used as key for our table lookup.

An example will help to understand the concept; in the figure bellow, we have the field "County Name" and the user has entered just one character, the letter "A". If the lookup table doesn't include a prefix, it would list all the counties starting with the letter "A". However, the lookup table has a prefix (the mnemonic code for the state) and thus, CSProX concatenates the mnemonic code ("CA") and the letter "A" entered to create the search key. The search key at the time was "CAA" and therefore, it lists only the California counties that start with the letter "A".



GETVALUE Function

Format:

Numeric-Variable-Name = GETVALUE(Alpha-Expression, Occurrence-Number);

Description:

This function returns the content of the numeric variable pointed by the alpha-expression passed as first parameter.

Example:

Let's assume that:

- Content of 'A' is 'Q1'
- Content of 'Q1' is 987 (Q1 is numeric variable)

Then, after the following statement is executed

V_Name = GETVALUE(A, 1);

The value of 'V_Name' is 987

GETAVALUE Function

Format:

Alpha-Variable-Name = GETAVALUE(Alpha-Expression, Occurrence-Number);

Description:

This function returns the content (string) of the alpha variable pointed by the alpha-expression passed as first parameter.

Example:

Let's assume that:

- Content of 'A' is 'B'
- Content of 'B' is "Q110" (B is an alpha variable)

Then, after the following statement is executed

V_Name = GETAVALUE(A, 1);

The value of 'V_Name' is "Q110"

GETDATE Function

This function calculates the date based on a referenced date and a period in days. The period can be positive, meaning that the period is counted forward or negative, meaning that the period is counted from the referenced date backward. The referenced date is passed as an eight digit integer number as follows: "YYYYMMDD" where YYYY is the year, MM the month and DD the day.

Format:

N = **GetDate**(Ref-Date, N-Days);

Where:

Ref-Date: is the date from which the period in days is computed (second parameter).

N-Days: is the number of days before or after the referenced date applied to compute the new date. If the period is positive, the date is supposed to be after the referenced date; if the period is negative, the date will be prior to the referenced date.

The function returns 0: if the referenced date is wrong (day and month are inconsistent or before Christ).

An eight digit number ("YYYYMMDD") with the date using the same format used to pass the relative date.

Often, the relative date will be the system date in which case the function can be invoked as follows:

```
Desired_Date = GetDate( sysdate(), N );

Desired_Date = GetDate( Year*10000 + Month*100 + Day, N );
```

GETTEXT Function

Format:

```
Alpha-Var = gettext( file_name, Key-Word[, key_len | Alpha_Char] );
```

[ ] indicates that this part is optional.

| indicates that one or the other keyword may be used

This function will retrieve from the file identified by ***file_name*** the record/string identified by the contents of alpha variable ***Key_Word***. The file key can be of fix length and use the first part of the record, in which case, the length is specified by *key_len.* The key can also be of variable length and the *key_len* parameter omitted, in which case, the key needs to be followed by a delimiter specified by ***Alpha_Char***. If both, the key length and the delimiter are omitted, the system will assume by default that the "=" sign is being used as delimiter.

The function is oriented to retrieve the string of characters that follows (i) the delimiter sign or (ii) the key component of the record when the key length has been specified.

No index file is used; if the key is not found, a null string is returned; if the key is found, the function returns the string following the delimiter sign or the key length and stored in **Alpha-Var**.

The file is loaded into memory the first time the function is executed and is refreshed or updated whenever changes are performed to the disk file. Thus, the memory file is always up-to-date regarding the disk file.

INITIALIZATION of Variables

Working Variables initialization is allowed at the declaration time.

Example:

```
Numeric Start_Up = FALSE, A = 1;
Alpha(8) Initial_Date = "20040511";
```

LOADCASE Function Extension

Format:

b = loadcase(ext-dict-name, var-list) [LOCK[(K)]];

Description:

The loadcase function reads a specified case from an external file into memory. Once the case is loaded, all variables defined in the corresponding external dictionary are available for use.

The "ext-dict-name" must be supplied. It is the internal name of the dictionary defined in the application for the external file.

The "var-list" specifies the list of variables that will identify the case to load from the external file. Note that "var-list" is not optional as opposed to the LOADCASE for local files. This process is similar to matching files on the basis of key variables in statistical and database software packages. Each of the variables in "var-list" must be defined in a dictionary or working storage. The combined length of the variables in "var-list" must equal the length of the case IDs defined for the external dictionary.

The optional "LOCK" parameter is used when the file related to "ext-dict-name" is remote and the case being retrieved needs to be blocked while is modified. While the case is locked no other user can have access to it.

The "K" parameter is also optional and is used to indicate the number of seconds the case needs to be blocked; the possible "K" values are:

-1: The Server Default value should be used

0: The case is blocked until the user frees it by another i/o operation (WRITECASE)

1-n: The number of seconds being blocked

The loadcase function concatenates the variables in the "var-list" to form a string. It then loads from the external file the case whose case identifier matches the string constructed from "var-list." If no "var-list" is provided, the next logical case in the external file will be loaded. The next logical case is defined as the case with the next sequential case identifier (in ascending order). This will not necessarily be the next case in physical sequence in the file.

Return value:

The function returns a value 1 (true) if the case was loaded successfully, 0 (false) otherwise.

Example:

```
OK = loadcase(SAMPDICT,CLUSTER,HH) LOCK(5);
```

The case, if successfully loaded, will be locked for 5 seconds or until an i/o operation is performed with the same case (whichever comes first).

LOCAL VARIABLES in user-defined functions

User-defined functions can define local variables that even having the same name as global variables are unique and have there own values while the function is executed. This is an important feature to prevent the crashing of global variables being modified by functions commands.

A simple example illustrates this feature:

```
FUNCTION MYFUNCTION( Alpha(32) File_Name )
        Numeric i;
        i = pos(".PFF", File_Name);
…
…
end;

…
…
While i <= n do
  X = MYFUNCTION( File_Name1 );

Enddo;
```

Both segments of the application are referencing the variable "i" however they are indeed two different variables and the do-loop won't be affected by the function.

MAKESHORTCUT Function

Format:

R = MakeShortCut([Title[, Work-Fold[, Icon-File-Path]]]);

This function is used to create a shot-cut to facilitate the launching of a given application.

The three function parameters can be omitted as shown on the function format and as explained below:

Title: is an alpha expression resulting in the short-cut title (normally a literal string). If omitted, the name of the application is used as title.

Work-Fold: is the folder name where the various application files are stored. If omitted, the current folder is used as working folder.

Icon-File-Path: is the specification of the complete icon file path. If omitted, the default CSEntry or CSBatch icon (depending on the application type) is used.

The return codes are as follows:

0: if the function is successfully executed and the short-cut is created.

1-4: Internal system errors.

5: The specified working folder could not be found.

On_Focus Function(*)

```
Function On_Focus()
….
end;
```

This function –if defined- is called automatically by the system each time a field focus is gained. The field focus is gained by: (i) pressing the <Tab> or <Enter> key –moving forward to the next field-; (ii) pressing the <Shift><Tab> keys –moving backward to the preceding field, if any-; (iii) using the mouse to move forward or backward to a specific field; (iv) by using the skip or reenter commands in the logic.

Although the function name is a system reserved name, the body of the function is entirely defined by the application developer according to the specific needs.

SELCASE Function Extension

Format:

N = **SELCASE**(Dict-Name, Key-Prefix, Offset) [**INCLUDE**(Var-Include-List)] [**MULTIPLE**] [LOCK[(K)]]

The SELCASE function retrieves all the keys from the data file associated with the data dictionary "Dict-Name" that starts with the prefix specified in "Key-Prefix". If the

MULTIPLE keyword is omitted, only one case can be retrieved –by simply clicking on the key desired-. On the other hand, if the MULTIPLE keyword is used, multiple keys can be marked for subsequent case retrieval using the **FOR** command as it will be explained below.

To mark one of the displayed keys, just click on the desired key; for subsequent selections, just hold the <Ctrl> key pressed and click on the following keys. To mark a group of consecutives keys, just click on the first one and then press the <Shift> key and click on the last one. If you want to unmark one of a set of marked keys, just press the <Ctrl> key and click on the key that needs to be unmarked. To unmark all the marked keys, just click on any one key.

Once all keys have been marked, pressing the <Enter> key or clicking on the "Accept Selection" icon (√) will create an internal list for subsequent cases retrieval. The FOR command is used to cycle through the key list, key by key, retrieving the corresponding or associated case. The case retrieval is done automatically by the system for each cycle of the FOR command. The syntax of the FOR is as follows:

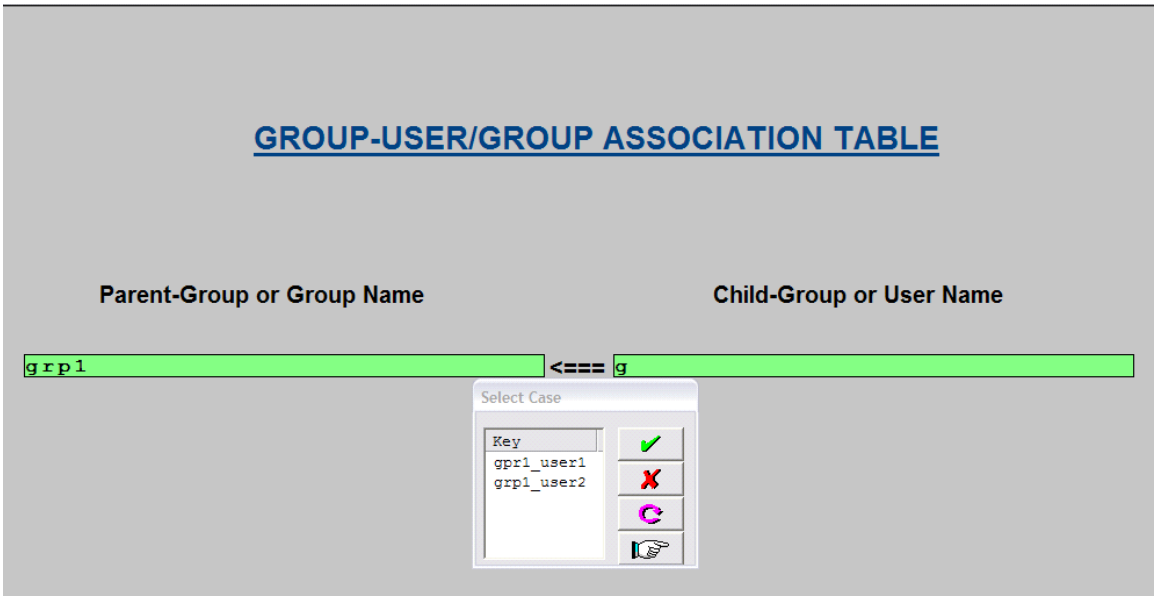
```
FOR Dict-Name DO
.....
.....
ENDDO;
```

"Dict-Name" is the Data Dictionary name associated to the data file that is being retrieved.

The "LOCK" keyword is used when the case(s) retrieved need to be locked while they are being modified by the application. "K" is the number of seconds that the case has to be locked. "K" can be:

-1:	tales the Default number of seconds defined in the server;
0:	the case will be locked until another i/o operation is performed on the same case (WRITECASE);
1-n:	the number of seconds.

The following image shows the dialog box of the SELCASE with MULTIPLE selections.



The dialog box above is displayed after the <Enter> key has been pressed in the "Child-Group or User Name" field. Since the "g" prefix was entered on this field, all keys that have the same prefix were retrieved and displayed on the dialog box. At this point, a subset or all the keys displayed can be marked to be retrieved –as oppose to the normal/single SELCASE where only one key can be retrieved simultaneously-. In this particular case, both keys can be marked and retrieved.

SET AUTOCODE command

This command is used in conjunction with the AUTOCODE function only.

Format:

SET AUTOCODE k;

"k" is a numeric constant defining the upper limit or maximum number of alternatives that will be displayed on the AUTOCODE dialog box before declaring the code description submitted to the matching process as vague. If the number of alternatives rendered by the matching process is higher than "k", the AUTOCODE function will return a "-3" code. At this point, the operator needs to refine the code description to come up with a more precise or meaningful text.

In the absence of the SET AUTOCODE command the default number or constant is used (10). Thus, the SET command is only needed if you want a different number of maximum options to be displayed.

SETVALUESET function(*)

This function allows the application developer to dynamically define or set a value set for the system range checking. This is a unique feature to constrain the values that can be entered in any field either using one of the various value sets defined for any field/variable or to dynamically define the ranges using information previously entered to further constrain the possible values.

An example should help to visualize the usefulness of this system function: Let's assume that in the children's section of a questionnaire there is one question that links the current child to the mother's line –back in the very beginning of the questionnaire, where the personal information of each household member was entered-. Normally, the range defined at the time the application is developed has to be 1-N, where N is the maximum number of household members that can be accepted. This is obviously a very broad range since very few households will reach this maximum. Therefore, in run time, this broad range can be improved or constrained in more than one way: first, the upper limit could be changed to the maximum number of persons of the current household. Nonetheless, this is still a broad range since we are talking about the mother and thus, only females are acceptable. Finally, a further range refinement can be done by restricting the possible mother candidates to those that have at least D years more than the child.

As it's easy to figure out, this function is specially CAPI oriented since if a contradiction between the answer and the application assertion about the feasible ranges arise, it can be solved immediately. In a data entry application, the methodology to solve the inconsistency might not be that simple and we could delay the operation by stopping the data entry process.

The function format is as follows:

1. R = SETVALUESET(*field-name*, *value-set-name*);
2. R = SETVALUESET(*field-name, array-name, alpha-array*);

The first format simply associate to a given field a value set. This can be a value set of the same field/variable (*field-name*) or a value set of a different variable but of the same type. Normally, the system range check uses the first value set defined; after the SETVALUESET function is executed, the *value-set-name* is used to perform the range check. Note that the function has to be executed either in the PREPROC or ONFOCUS procedure of *field-name* or before.

The second format uses two arrays to dynamically define the ranges for *field-name*. The first array (*array-name*) is either a numeric or alpha array depending on the type of *field-name*. If *field-name* is numeric, *array-name* has to be numeric too. If *field-name* is an alpha field, *array-name* has to be alpha-numeric too. *Alpha-array* is used to define the corresponding code labels defined in *array-name*. Only a discrete value set can be defined using this format. Version 2 of CSProX will allow for the definition of intervals too.

Note that arrays are zero based (the first element is in position 0 and not 1). Thus, when format 2 is used, codes and labels have to be stored starting in position 0. The system will look for the proper value from position 0 and up to the first NOTAPPL or BLANK value –depending on the *array-name* type- stopping as soon as the value is found.

Example: Let's say that in the first module of a household questionnaire we have a roster like

the one shown below, and one of the last modules is a children module where among other information, a link with the mother's child is established. The range as defined in the DD is 1-20, being 20 the maximum number of household members and/or the number of occurrences of the record/roster.

The following instructions are used to narrow down the range selection at the time the mother's line number has to be entered:

```
{1}  clean_labels();
{2}  n = count( roster_r1 where SEX = 2 and (R1_Age-R2_Age)>14 set idx );
{3}  i = 1;
{4}  while i <= n do
{5}    ACodes(i) = idx(i-1);
{6}    ALabels(i) = concat( strip(r1_first_name (idx(i-1))), " ", strip(r1_last_name(idx(i-1))) );
{7}     i = i + 1;
{8}  enddo;
{9}  setvalueset(Mother_Line, ACodes, ALabels)
```

1. The function "clean_labels" clears previous values of both arrays ACodes and ALabels!! (user-defined function)
2. Counts the number of mother candidates: SEX=2 and Age Difference > 14. In addition, all positions (equivalent to line number) of household members satisfying the condition are stored in the array IDX (see COUNT function extension).

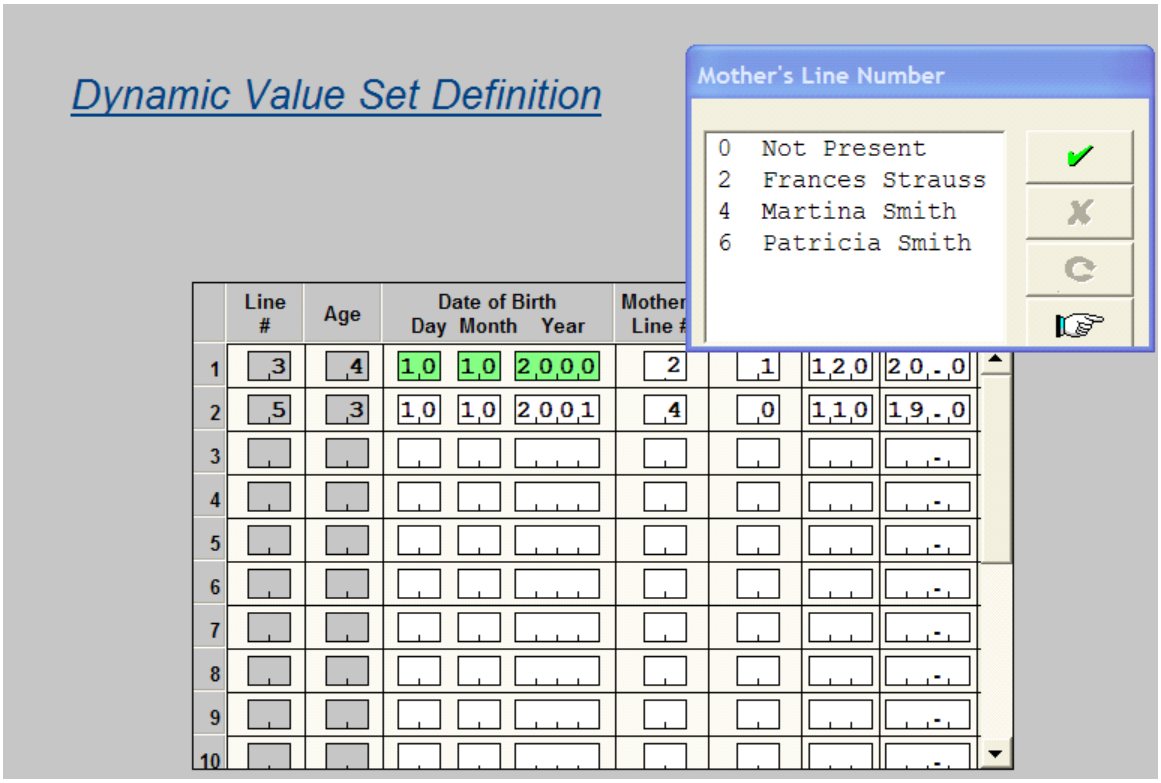
	Line #	Name	Last Name	Respondent	Sex	Age	Relationship
1	1	John	Strauss	0	1	48	1
2	2	Frances	Strauss	1	2	45	2
3	3	Frank	Strauss	0	1	4	3
4	4	Martina	Smith	0	2	29	7
5	5	Helen	Johnson	0	2	3	7
6	6	Patricia	Smith	0	2	24	7
7							
8							
9							
10							
11							

3-8. The do-loop copies line numbers from the IDX array to ACodes and the full name to ALabels.

9. Finally, the SETVALUESET function sets both arrays as the categories and labels to be used in the "Mother_Line" field, producing the CAPI dialog box shown in the figure below.

The dialog box shows 3=three possible candidates plus the option "Not Present". Thus, the respondent's answer should be one of these. Any other response would be in conflict with the

information gathered in the first roster (or the rule to select the mother candidate should be relaxed).



SETAVALUE Function

This function is used to define an alpha variable pointed by the content of an alpha expression (first parameter) by another alpha expression (third parameter). The second parameter is the occurrence number of the variable being defined. If the variable is single, a 1 needs to be specified.

Format:

ret = SETAVALUE(Alpha-Expression-1, Occurrence-Number, Alpha-Expression-2);

Where:

Alpha-Expression-1 should provide the name of a variable where the name of the target variable to be defined should be stored.

Occurrence-Number: It should be the occurrence number of the target variable; if the target variable is single, this parameter should be 1.

Alpha-Expression-2: It should be the string that will be assigned to the target variable.

The function return:

1 if the operation is successful and 0 otherwise.

0 if the operation didn't succeed.

Example:

```
PROC Q110
    ret = SETAVALUE(  A, 1, getsymbol() );
```

Let's assume that the value of the alpha variable 'A' is 'B'. Then, after the function is executed, the variable 'B' will have the value "Q110".

SETVALUE Function

This function is used to define a numeric variable pointed by the content of an alpha expression (first parameter) by a numeric expression (third parameter). The second parameter is the occurrence number of the variable being defined. If the variable is single, a 1 needs to be specified.

Format:

```
ret = SETVALUE( Alpha-Expression, Occurrence-Number, Numeric-Expression );
```

Where:

Alpha-Expression should provide the name of a numeric variable where the name of the target variable to be defined should be stored.

Occurrence-Number: It should be the occurrence number of the target variable; if the target variable is single, this parameter should be 1.

Numeric-Expression: The result of this expression will be assigned to the target variable.

The function return:

1 if the operation is successful and 0 otherwise.

0 if the operation didn't succeed.

Example:

```
ret = SETVALUE( A, 1, n+1 );
```

If the content of the variable "A" is "B" being B a numeric variable, the above instruction would be equivalent to: "B = n+1".

SHOW Function Extension

The show function permits to display on the screen a set of variables with their corresponding values in a simple and organized fashion. It's a convenient way to define dynamic help screens in CAPI applications that can be invoked at any time during the interview.

Format:

N = Show([Show-Name] [Group-Name | Relation-Name] Object-List,

[Title(Title-List)]);

Where:

Show-Name: is an alphanumeric constant or literal/character string with the name of the help box. This alpha constant will be displayed at the top of the show box as a way to individualize the help screen (e.g. "Household Structure").

Group-Name is the name of a repeating or single group.

Relation-Name is the name of an existing relation.

Object_List = [Var-List] [, function-name(Var-List)] [,Var-List]

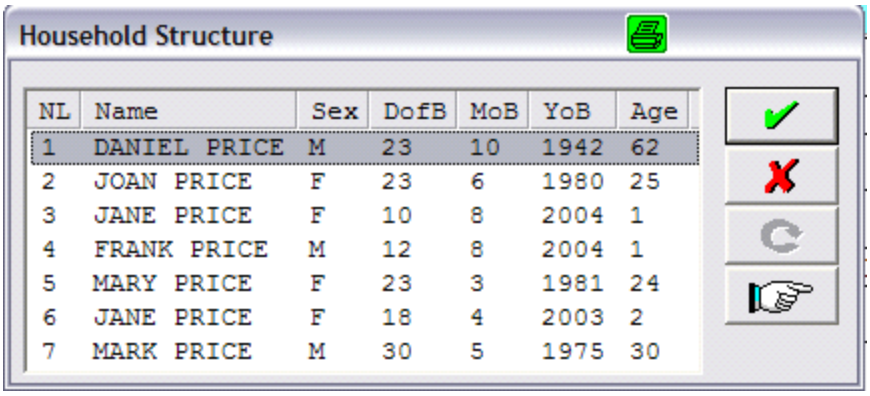
Var-List = Var-Name[, Var-List]

Var-Name = Input-Var-Name | Work-Var-Name

The following is an example of a help/show screen from the Demo application:

```
n = show( "Household Structure" M1A_R, M1_Q01, concat(strip(M1_Q02), " ", strip(M1_Q03)),
     alphasex( M1_Q04 ), M1_Q05D, M1_Q05M, M1_Q05Y, M1_Q06,
    title("NL", "Name", "Sex", "DofB" "MoB" "YoB" "Age") );
```

Displaying the following screen:



In the above example, we are using three functions: the first and second one, are system functions to concatenate the first and the last names; the third one is a user's function to convert the sex code from numeric to alpha (1->M and 2->F). The result is what you can see on the figure above.

SYSINFO Function

Format:

n = sysinfo(InEntry | InAdvance | InBatch | EntryMode | Path | Mouse | Event | Forward | Backward);

| indicates that one or the other keyword may be used

This function provides information about CSProX operation modes (CSEntry and CSBatch) allowing the application developer to check for system status and other overall information at execution time.

- N = **sysinfo**(**InEntry**);

The function returns TRUE (1) if the system is in ENTRY mode (ADD, MODIFY or VERIFY) and FALSE if it is running in Batch.

- N = **sysinfo**(**InBatch**);

The function returns TRUE (1) if the system is executing the Batch driver FALSE (0) if it is running the Entry driver.

- N = **sysinfo**(**InAdvance**);

The function returns FALSE (0) if the system is not in ADVANCE mode and TRUE (1) if it is in ADVANCE mode. The ADVANCE mode is set when <F6> or <F10> is pressed or by a mouse click on a field that is beyond the current field (forward).

- N = **sysinfo**(**EntryMode**);

The function returns 1:ADD, 2:MODIFY, 3:VERIFY and 0 if is not running in ENTRY mode.

- N = **sysinfo**(**Path**);

The function returns 0 if CSEntry is running in PATH OFF (operator controlled) and 1 if it is running in PATH ON (system controlled).

- N = **sysinfo**(**Mouse**);

The function returns 0 if the mouse is disable and 1 if it's enable.

- N = **sysinfo**(**Event**);

The function returns the code of the event is currently processing according to the following list: 1: PreProc; 2: OnFocus; 3: KillFocus; 4: PostProc; 5: OnOccChange.

- N = **sysinfo**(Forward);

The function returns TRUE (1) if the motion is FORWARD and FALSE (0) otherwise.

- N = **sysinfo**(Backward);

The function returns TRUE if the motion is BACKWARD and FALSE otherwise.

Three new events have been added to the Pre and Post procedures: the OnFocus event –procedure- is executed immediately after the PreProc whenever the flow motion is forward; on the other hand, if the motion is backward, the OnFocus procedure is executed immediately before the DE operator has access to the field (immediately before the interface gets the control back). The KillFocus procedure is executed right before the PostProc procedure is executed –if the motion is forward- or just before the OnFocus procedure of a preceding field is executed –if the motion is backward-. The OnOccChange procedure is executed at the time the number of occurrences of a roster is increased or decreased.

- N = **sysinfo**(**InBatch**);

The function returns 0 if the system is running in ENTRY mode and 1 if it's running in BATCH mode.

- N = **sysinfo**(**Cases**);

The function returns the number of input cases read in by the Batch processor at the time the function is executed.

- N = **sysinfo**(**ISWINCE**);

The function returns TRUE if the system is running under Windows CE and FALSE otherwise.

WRITECASE Function extension

Format:

```
b = writecase(ext-dict-name,var-list);
```

Description:

The **writecase** function writes a case from memory to an external data file. It can be used to update existing cases or to write new ones. If the data file is remote and the case identified by "*var-list*" was locked by the same user, it is **automatically unlocked**. If the case was locked by a different user, the WRITECASE fails and a "0 " code is returned (see below the Return Values).

The "ext-dict-name" must be supplied. It is the internal name of the dictionary defined in the application for the external file.

The optional "var-list" defines the case identifiers in the external file. The **writecase** function concatenates the variables specified in "var-list" to form a string whose length must be the same as the length of the case identifier in the external dictionary. All variables in the "var-list" must exist in a dictionary or working storage. If no "var-list" is provided, the current values of the identifiers in memory for the external file are used.

If the case identified by "var-list" already exists, the **writecase** function will overwrite the existing case. The **writecase** function automatically generates and updates the index file (with extension IDX) for the external data file.

After a case is written to an external file, the external dictionary variables for that case remain in memory. If the application does not assign new values to all variables in the external dictionary before the next **writecase** function is executed, then values from the previous case will be written to the external data file. Use the clear function to clear the values of these variables.

Return value:

The function returns a logical value of 1 (true) if the write is successful and 0 (false) otherwise.

Example:

```
OK = writecase(KIDS,CLUSNUM,HHNUM,LINE);
```

ARABIC Data Dictionary

Description:

The data dictionary now can be defined in Arabic language. The Dictionary, Level, Record, Item and value sets labels can be defined in Arabic.

ARABIC support in WINCE

Description:

There is partial-Arabic support in the Pocket PC version (notes, messages, questions and value-set can be displayed in Arabic)

MATCH Function

Format:

b = exec("_match", *RegularExpresion, AlphaExpresion*);

Return value:

Return 1 if AlphaExpresion match with RegularExpresion. Otherwise returns 0.

Example:

Check valid e-mail address (at least 1 character before @, at least 1 character after @, finish with "." plus 2,3 or 4 letters

```
if exec( "_match", "^([_a-z0-9-]+)(\.[_a-z0-9-]+)*@([a-z0-9-]+)(\.[a-z0-9-]+)*(\.[a-z][a-z][a-z]?[a-z]?)$", MailAddress )=1
```

Enabling/Disabling messages in the listing file

Format:

set behavior() MsgNumber display off|on,

Description:

This command allows to enable or disable messages

Example:

Run a batch application but the message 9815 will not be written to the listing file when runmode is 2.

```
if runmode=2 then
   set behavior() messages(   9815   )  off;
else
   set behavior() messages(   9815   )  on;
endif
```

Note : To disable system messages, use the corresponding negative number

GETLABEL function extension

Format:

Alpha = getlabel(VarName/ValueSet BY INDEX, NumericExpr) ;

Description:

This function gets the code (in ASCII) associated to the line number of the value set using the relative position.

Return value:

The corresponding ASCII code from the value-set specified. When *VarName* is used as parameter, the current value set is used.

Example:

```
[ValueSet]
Label=List of crops
Name=R5_Q002_VS1
Value=200;CASH CROPS
Value=201;... Maize
Value=202;... Cotton
Value=203;... Other (specify)
Value=210;VEGETABLES
Value=211;... Beans
Value=212;... Potatotes
Value=213;... Pumpkins
Value=214;... Groundnuts
Value=215;... Jugo beans(tindlubu)
Value=216;... Cabbage
```

To return the product codes:

```
getlabel( R5_Q002 by index, 1 )  → returns "200"
getlabel( R5_Q002 by index, 2 )  → returns "201"
getlabel( R5_Q002 by index, 11) → returns "216"
getlabel( R5_Q002 by index, 12) → returns ""
```

To fill an array (alist) with the corresponding codes and labels do the following:

```
n = 11;
do i = 0 while i < n
  code = tonumber(getlabel( R5_Q002 by index, i+1 ));
  alist(i) = getlabel(R5_Q002, code );
enddo;
```

ONSYSERROR function

Format:

function OnSysError(*iMsgNum*, *alpha(64) msgText*)

Description:

OnSysError is called automatically when a system message is produced (for example: 88888 Out of range, 92101 Case ids ' ' duplicate and existing case, etc....).

OnSysError works in Entry and Batch applications

Example:

```
{Write to an external file the messages)
function OnSysError( iMsgNum, alpha(64) msgText )
        nMsg  = nMsg + 1;
        EXTERNAL_ID = nMsg;
        EXTERNAL_MSGNUM= iMsgNum
        EXTERNAL_MSGTEXT= MsgText
        WRITECASE( EXTERNAL_MESSAGES, EXTERNAL_ID  );
end;
```

Note: In batch mode when OnSysError return 0, the message will not be written to the listing file.

Conditional compilation

Format:

```
{<EnvVar
        ...this code compile only when 'EnvVar' has been defined
EnvVar>}
```

Description:

The system recognizes the following symbols: "{<" and ">}"

EnvVar is an environment variable. The corresponding block is compiled only if EnvVar

has been defined.

In CSPro public the block is not compiled because it's treated as a comment.

Example:

```
{<SENEGAL
        … your code for Senegal
SENEGAL>}

{<ZAMBIA
        … your code for Zambia

ZAMBIA >}
```

PORTRAIT/LANDSCAPE Functions (WINCE only)

Format:

PORTRAIT()
LANSCAPE()

Description:

Set the PPC screen to portrait or landscape mode.

GPS Function

Format:

GPS(OPEN/CLOSE)
GPS(GpsInformation, LastMeasureTime)

Description:

This function provides information from the GPS. The parameters OPEN/CLOSE are used to open and close the GPS.

'GpsInformation' is a numeric array (4 elements) and 'LastMeasureTime' specify the maximum amount of seconds to allow as valid the information provided by the GPS. If the information provided by the GPS is older than 'LastMeasureTime' seconds ago, the function will return 0 (false).

The following information is returned in NumericGpsArray (positions 0 to 3):

- Latitude
- Longitude

- Altitude
- Number of satellites.

Note (WIN32 version): when a GPS is not found, the GPS(OPEN) function will scan the ports COM1 to COM20 with different speeds (4800, 9600,19200 and 2400) searching a valid GPS. If the GPS is found, the Port number and Speed are saved in the Windows Registry (HKEY_CURRENT_USER\Software\Serpro S.A.\CSPro 4.1 Data Entry\Settings: GPSPort and GPSBaudRate), so the next time that information is extracted from the registry. This is a very important feature because the GPS could be installed in different ports in different machines and the CSProX program will be the same.

GETXLS Function

Format:

GetXls(ExcelFile, ExcelSheet, ItemList);
GetXls(ExcelFile, ExcelSheet) REWIND;

Description:

This function loads information directly from an excel file or a comma delimited file.

ExcelFile is a '.XLS' file. *ExcelSheet* is specific sheet from the *ExcelFile*. *ItemList* is a list of items (numeric or alpha) defined in some dictionary or declared in the logic.

Return value:
The function returns the number of columns loaded or -1 when there is no more information to read (there is no more rows).

Example:

```
e = GetXls( "IDENTIFICATIONAN.xls", "!Identificationan",
    Q003,Q005,Q006,Q007,Q008Day,Q008Month, Region );
```

This function read from the file IDENTIFICATIONAN.xls (sheet Identificationan) the columns Q002 until.Region.

The excel sheet can have more columns but the system will read only the columns specified in the function.

To start again from the beginning:

```
e = GetXls( "IDENTIFICATIONAN.xls", "!Identificationan" ) REWIND;
```

Occurrence Labels

The data dictionary allows occurrence labels for records having multiple occurrences.

The occurrence labels can be used in Frequency, Export and with the *Getlabel* function.

Additionally, the CSPro designer allows the use of "%s" within the roster stub in order to retrieve and display the label associated to the occurrence.

In the examples below, the labels: "HIV counseling", "HIV counseling for pregnant women", ..., "HIV/AIDS preventive outreach services" are defined in the occurrence label of the corresponding record.

Example1: Frequency

Item R2_Q210: Charges fee, occurrence 1(HIV counselling)
... tbd-name: '.FACILITY_DICT.R2A3.R2_Q210'

…

Item R2_Q210: Charges fee, occurrence 2(HIV couselling for pregnant women)
... tbd-name: '.FACILITY_DICT.R2A3.R2_Q210'

……
…....
Item R2_Q210: Charges fee, occurrence 9(HIV/AIDS preventive outreach services)

Example 2: Export to SPSS

/R2_Q210$1 "Charges fee: HIV counselling"
/R2_Q210$2 "Charges fee: HIV couselling for pregnant women"
…..
…..
/R2_Q210$9 "Charges fee: HIV/AIDS preventive outreach services"

Rosters can include items from different records

Description:

Rosters accept items from different records (or the entire records) as long as they have the same number of occurrences (parallel records).

Exclude elements from a ValueSet

Format:

SetValueSet(....) EXCLUDE(VarList);

Description:

VarList can contain single or multiple items. In runtime every item is evaluated from the VarList and those codes are excluded from the value-set.

Example:

Suppose we have a list of products and we need to rank the 3 most important starting with the most important to the less important. We would define a multiple item (for example: R5_Q002) with 3 occurrences and we would use the exclude clause in *SetValueset* to exclude from the value set dynamically the codes entered on the previous occurrences.

```
[ValueSet]
Label=List of crops
Name=R5_Q002_VS1
Value=200;CASH CROPS
Value=201;... Maize
Value=202;... Cotton
Value=203;... Other (specify)
Value=210;VEGETABLES
Value=211;... Beans
Value=212;... Potatotes
Value=213;... Pumpkins
Value=214;... Groundnuts
Value=215;... Jugo beans(tindlubu)
Value=216;... Cabbage
…
…
```

```
PROC R5_Q002
OnFocus
  setvalueset( R5_Q002, R5_Q002_VS1 ) exclude( $ );
```

On the first occurrence we will see the full value set list. If code **211** is selected on the first occurrence, when the second occurrence is reached, the list will automatically exclude it showing the remaining crops. Similarly, if code **216** is selected on the second occurrence, when the third occurrence is reached, both codes -211 and 216- will be excluded from the value set list displayed.

Numeric fields grouping in Entry

Description:

When a numeric field gets the focus, the number entered is visually split into 3 digit groups as illustrated in the example below. Each group is delimited in two different ways: by the vertical line separator and by an alternate color used to highlight the number magnitude.

Example:

CLEAR Function extension

Format:

CLEAR(WorkingDictionary)

Description:

The CLEAR function can be used with Working dictionaries.

ALIAS command

Format:

ALIAS *DictionayElement*, *AliasName* [, *DictionayElement*, *AliasName*,[...]];

Description:

The ALIAS command is normally applied to frequently used items/variables to make programming easier.

Example:

ALIAS R1A_Q010, **age**, R1A_Q011, **sex**;

It's possible to use 'age' or 'R1A_Q010' as synonyms in the CSPro application.

"if R1A_Q010, in 15:49 and R1A_Q011=Female then"

is equivalent to:

"if **age** in 15:49 and **sex**=Female then"

GETMACHINEID Function

Format:

Alpha = GetMachineId();

Return value:

The function Returns an alpha string with the internal machine id (works on WIN32 and PPC). The purpose behind this function is to be able to identify either a PC or PPC with a given list of ids either for security reasons or other purposes. If data are extracted from a PPC, the id could be used to create data folder names of each interviewer of a given team.

GETPATH Function

Format:

exec("_getpath", AlphaVar);

Description:

It gets the full CSPro path and stores it in *AlphaVar.*

Example:

```
alpha(128) CsProPath ;

exec( "_getpath", CsProPath );
```

GETCURDIR Function

Format:

exec("_getcurdir", AlphaVar);

Description:

Gets the current directory path storing it in *AlphaVar*

Example:

```
alpha(128) CurDir;

exec( "_getcurdir", CurDir );
```

SPLIT Function

Format:

nParts = exec("_split", *OutputArray*, *InputString*, *Separators* [*ExclusionArray*]);

Description:

The split function takes the *InputString* (alpha expression) and split it in different words/sentences based on the separator specified in the *Separators* parameter (alpha expression). The function returns the number of words and each word is saved on the *OutputArray* (alpha array). It's possible to specify an ExclusionArray (alpha array) to exclude some specific words from the output.

Example:

```
numeric i,n;
array alpha(20) aOutput(20);
array alpha(20) aExclude(10);

aExclude(0) = "is";
aExclude(1) = "a";
aExclude(2) = "of";

 n = exec( "_split upper upper", aOutput, "This is a Test of;split , function", " ;,", aExclude);

 write( "Split: n=%d", n );
  i = 0;
  while i < n do
          write( "%d)=(%s)", i,  aOutput,(i) );
          i = i + 1;
 enddo;
```

The output will be:

```
Split: n=4
0)=(This)
1)=(Test)
2)=(split)
3)=(function)
```

ASSERT0 Function

Format:

ASSERT0(*logical_expresion*);

Description:

The function generates an error message when *logical_expresion* is FALSE.

Example:

ASSERT0(Sex = Female);

ASSERT0 will generate an error message when Sex is different from Female.

TRACE Function

Format:

TRACE(Parameters);

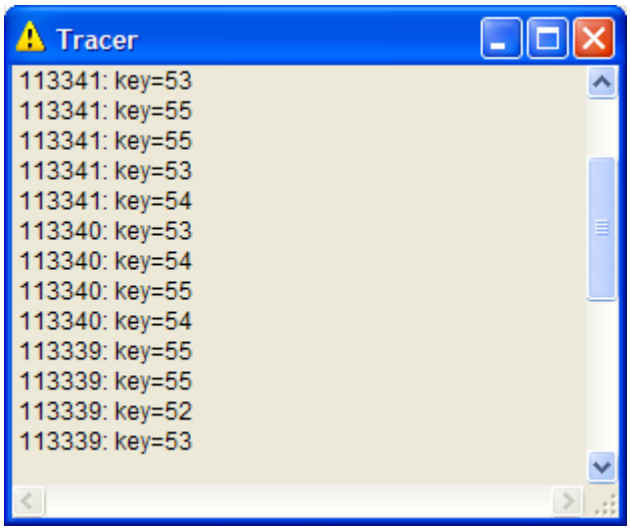
See the errmsg function for the parameters description.

Description:

The function is similar to the errmsg function with the following differences: (1) the message is displayed on a special window at the top left corner of the screen as illustrated on the figure below; (2) unlike the errmsg, this function doesn't interrupt the processing flow; (3) the window can be cancelled at any time.

Example:

```
function OnKey( k )
  trace( "%t: key=%d", systime( "HHMMSS"), k );
  OnKey = k;
end;
```



ENCRYPT/DECRYPT Functions

Format:

```
Alpha=ENCRYPT( AlphaExprToEncrypt, Key);
Alpha=DECRYPT( AlphaExprToDecrypt, Key);
```

Description:

These functions are used to encrypt/decrypt strings using a 256-bit AES encryption algorithm.

Return value:

The functions return an alpha string containing the encrypted or decrypted text.

Example:

```
PROC GLOBAL
  alpha(64) thePwd="My password";
  alpha(64) theKey="My Encription key";
  alpha(64) EncryptedPwd;
  alpha(64) DecryptedPwd;

PROC TEST_FF
   EncryptedPwd = encrypt( thePwd, theKey );
   DecryptedPwd = decrypt( EncryptedPwd,theKey );

    write( "Pwd='%t', Enc'%t', Dec='%t'", thePwd, EncryptedPwd, DecryptedPwd );
```

The output generated will be:

```
Pwd='My password', Enc'wrN4uGtgG6aqVMVdbR+Cew==', Dec='My password'
```

LOGIN/LOGOUT Functions

Format:

```
b = LOGIN() ;
b = LOGOUT() ;
```

Description:

These functions allow to login and logout from the CSProX server. Those functions are used by CSEntry.

LOGINNAME Function

Format:

Alpha= LOGINNAME();

Return value:

This function returns an alpha string containing the current login name connected to the CSProX server.

SPECIALZERO Function

Format:

SPECIALZERO(TRUE/FALSE);

Description:

This function set/unset an internal flag to interpret the special values (missing or notappl) as zero in the 4 basic arithmetic operations: +, -, * and /.

Example:

Suppose that A2 and A3 are skipped fields(notappl) and A1=10 and A4=23. When you make the summation “A1+A2+A3+A4” you will get NOTAPPL as result. You can avoid that using the specialzero function.

```
PROC GLOBAL
numeric a1=10,a2=notappl,a3=notappl,a4=23;
numeric total;

PROC TEST_FF
   total = a1+a2+a3+a4;
   write( "total1=%d", total );

   specialzero(true);
   total = a1+a2+a3+a4;
   write( "total2=%d", total );
   specialzero(false);

   total = a1+a2+a3+a4;
  write( "total3=%d", total );
```

The output generated will be:

```
total1=NOTAPPL
total2=33
total3=NOTAPPL
```

Question file compilation

Description:

The QSF file is compiled to check that the parameters imbedded in the questions have been properly defined in the application. The parameters must be variables names or user functions. The system produces an error message when a parameter is not found in the dictionary, the working area or the user's functions definitions.

Questions with user functions parameters

Description:

The QSF file allows user functions as parameters.

Example:

In the dictionary we have an item from a multiple record....

```
[ValueSet]
Label=Items
Name=R9_Q032_VS1
Value=360;Car insurance premium
Value=361;Life insurance premium
Value=362;Local authority rates
Value=363;Subscriptions
Value=996;Other
Value=999;End of List
```

In the GLOBAL procedure we define the function that will return the product name associated to the current occurrence.....

```
function alpha(32) ProductName()
  ProductName = getlabel( R9_Q032, R9_Q032(curocc()) );
end;
```

In the QSF (R9_Q033 field) the 'ProductName' function can be used as parameter as follows:

What was the total cost of the %ProductName% in the last year?

Finally in runtime the R9_Q033 question will be expanded as....

What was the total cost of the 'insurance premium' in the last year? or
What was the total cost of the 'Life insurance premium' in the last year?, etc. depending on the value of R9_Q032

Truncating alpha expresions

Description:

A new parameter type has been added to the errmsg family functions: **%t**

%t matches with an alpha expression and it works as strip function.

Example:

```
errmsg( "First Name='%s', MidName='%s', Last Name='%s'",
 strip(FirstName), strip(MidName), strip(LastName) );
```

can be written as:

```
errmsg( "First Name='%t', MidName='%t', Last Name='%t'",
 FirstName, MidName, LastName );
```

EXEC menu extension

Description:

This function now allows an alphanumeric array as parameter.

Example:

```
PROC GLOBAL
ARRAY ALPHA(50) NamesList(50);

PROC IT1
NamesList(0) = “John”;
NamesList(1) = “Peter”;
NamesList(2) = “Mary”;
NamesList(3) = “William”;
NamesList(4) = “Patrick”;
NamesList(5) = “”;  {End of List}


e= exec( "_menu", “Select a Name”, NameList );
if e >= 0 then
        errmsg( “The person  selected is: ‘%t’”, NameList(e) );
endif;
```

OnPartialSave Function

Description:

This function permits absolute control on the partial save originated by the interviewer or by the logic. When the partial save is originated, the function ÖnPartialSave is executed and the application programmer has the chance to either reject the partial save, perform any type of control procedure, etc. This function operates in a similar manner as the OnKey function and the On_Focus function in the sense that they are called when some events are triggered rather than by a specific function call.

Extensions to CSPro Data Dictionaries

Two important extensions have been added to the CSPro data dictionaries that add even more power and flexibility to the software:

1. The ability to load and store data dictionaries using the DDI scheme. This ability allows the integration with other software that also uses the DDI schema which has become a standard in data documentation, portability and dissemination of data.

   Unfortunately, DDI has a new release (3.0) which is one level up of our implementation and we still haven't made the evaluation regarding the effort level to port it to the latest version.

2. An important effort has been carried out to transform the CSPro DD into a multi-language DD. Multiple languages are supported within one DD opening up a new horizon for the international statistical community. With a unique DD, they will be able to produce applications in different languages switching from one to another dynamically in a similar manner as the questions file in CAPI does. This is an excellent feature not only for CAPI but to produce tables in different languages, to export DDs and data to other statistically systems also in different languages.

Minor improvements

1. Reenter to protected fields are accepted
2. Balloon helper: used to provide field information (type, length, etc) on the forms screen of designer when the mouse is positioned over any field.
3. Skip to Next without target skips to the first variable of the next line.
4. On_RClick function: in entry, making right click on a field it executes the function passing as parameter the field name.
5. The FOR statement accept a WHERE clause as filter; only those cases with the WHERE clause equal TRUE will get in the FOR cycle.
6. The DELETE CASE button can be deactivated to inhibit the case deletion.