MODULE 7: C/AL FUNCTIONS

Module Overview

Sometimes identical code must be run from different locations. Other times a similar code with different variable values must be run from different locations. Putting such code in a function when it must be called from different locations saves development time, and streamlines managing and debugging the code.

C/AL provides many built-in functions that you can use in your code or from your objects. These functions are predefined to achieve certain tasks, such as performing a string transformation, retrieving a system date, and so on.

To successfully build custom Microsoft Dynamics NAV 2013 applications, you must be familiar with the built-in C/AL functions, and understand how to create custom functions.

Objectives

The objectives are:

- Explain the concepts of functions and parameters.
- Explain the C/AL Symbol Menu.
- Describe the use and syntax of data access, filtering, and manipulation functions.
- Describe the use and syntax of user interaction functions.
- Describe the use and syntax of string functions.
- Describe the use and syntax of system functions.
- Describe the use and syntax of date functions.
- Describe the use and syntax of number functions.
- Describe the use and syntax of array functions.
- Describe the use and syntax of several other important functions.
- Provide an overview of the benefits of creating custom functions.
- Explain the concepts of local functions and local variables.
- Create custom functions in a page and call the functions from Actions.

Functions and Parameters

Functions are a fundamental programming concept. A *function* is a named part of a program, also known as a subprogram or a subroutine. When code that is running reaches a function, the main application is paused while the function code is handled.

Lesson Objectives

Functions

When the function name, which is known as the identifier, is used, the current program is suspended and the trigger code for the specified function executes. Using the identifier in this manner "calls" the function. When the trigger code in the called function completes, the function "returns" to where it is called from. The way that the function is called determines what happens when it returns.

A function can be used in an expression.

For example, the following code uses a function named CalculatePrice in an expression:

Function call in an expression

```
TotalCost := Quantity * CalculatePrice;
```

The **CalculatePrice** function returns a value that is used in evaluating the expression. This return value then is multiplied by the *Quantity* variable. That result is assigned to the *TotalCost* variable.

You can call a function by using a function call statement. This statement only calls the function and returns no value.

The example shows how to call a function named **RunFunction** as a statement:

Function call statement

```
IF Quantity > 5 THEN
   RunFunction;
```

The RunFunction returns no data back to the calling trigger.

Parameter

A *parameter* is one or more variables or expressions that are sent to the function through the function call. The parameter provides information to the function, and the function can change that information. If a function has parameters, the function identifier has a set of parentheses that follows it. Within these parentheses are one or more parameters. If there is more than one parameter, the parameters are separated by commas.

The following example shows how to call a function that has two parameters:

Function call with parameters

```
CubeVolume := POWER(SideLength,3);
```

Parameters can be simple values or expressions. If an expression is used as a parameter, then it is evaluated before the value is passed to the function.

In the following example, use an expression as a parameter for the **Power** function:

Expression as a parameter

```
GrossVolume := POWER(SideLength + PackagingThickness * 2,3);
```

Pass by Value

When a parameter only sends a piece of information to a function, the parameter is passed *by value*. The parameter knows only the value of the variable or expression that is used for the parameter. Because it is only a value, any change that the function makes to this parameter does not affect any variables in the calling trigger.

Pass by Reference

When a parameter is passed to the function and the function changes that parameter, the parameter is *passed by reference*. The parameter knows the variable's location in the computer memory, and it passes the computer memory location to the new function. Any changes that the function makes to this kind of parameter are permanent and affect variables in the calling trigger.

If a parameter is passed by value, you can use any expression for that parameter. However, if a parameter is passed by reference, a variable must be used for that parameter so that its value can be changed. Variables have a location in memory, whereas expressions or constants do not.

***Note:** When you use text parameters by reference, the actual length of the parameter is the length of the variable being passed, instead of the declared length of the parameter. For example, if you pass a text variable of length 10 by reference to a text parameter of length 20, the actual **MAXSTRLEN** of the parameter is 10.*

Formal and Actual Parameters

A formal parameter is declared in the function definition. For example, the DELSTR function has the following syntax:

Code Example

```
NewString := DELSTR(String, Position [, Length])
```

The words *String*, *Position*, and *Length* are the formal parameters.

The actual parameter is used when the function is called.

The following example shows how to call the DELSTR function:

Calling DELSTR

```
UserInput := DELSTR(UserInput,Comma,1);
```

The variables *UserInput* and *Comma* and the constant 1 are the actual parameters.

Actual parameters always correspond to the formal parameters. The formal and actual parameters must match in the number, order, and data type. The actual parameter *UserInput* becomes the formal parameter *String*; the actual parameter *Comma* becomes the formal parameter *Position***;** and the actual parameter *1* becomes the formal parameter *Length*.

Because this code uses a pass-by value for all three parameters, the values of the three actual parameters are actually passed to the function. Therefore, the actual parameters are not changed when the formal parameters are changed inside the function.

If the code uses a pass-by reference instead, it passes the variable references (memory addresses) to the function. Then the actual parameters are changed if the corresponding formal parameters are changed inside the function. Because constants cannot be changed, 1 cannot be used as an actual parameter if Length is passed by reference.

Built-in Functions

A *built-in function* is provided by the C/AL language. Its trigger code cannot be viewed, because it is built into the programming language. The following table describes several built-in functions:

Function Name	Description
MESSAGE	Displays a message on the screen.
MAXSTRLEN	Returns the defined length of a string variable.
COPYSTR	Returns a part of a string.
CLEAR	Clears the passed-in variable.
ARRAYLEN	Returns the number of elements in an array.

Functions such as **MESSAGE** and **CLEAR**, have no return value. You can only call them through a function call statement. Functions such as **COPYSTR** and **MAXSTRLEN** return a value. You can use them in an expression. The **CLEAR** function changes the passed-in parameter. It is an example of *passing by reference*, whereas the other functions are examples of *passing by value*.

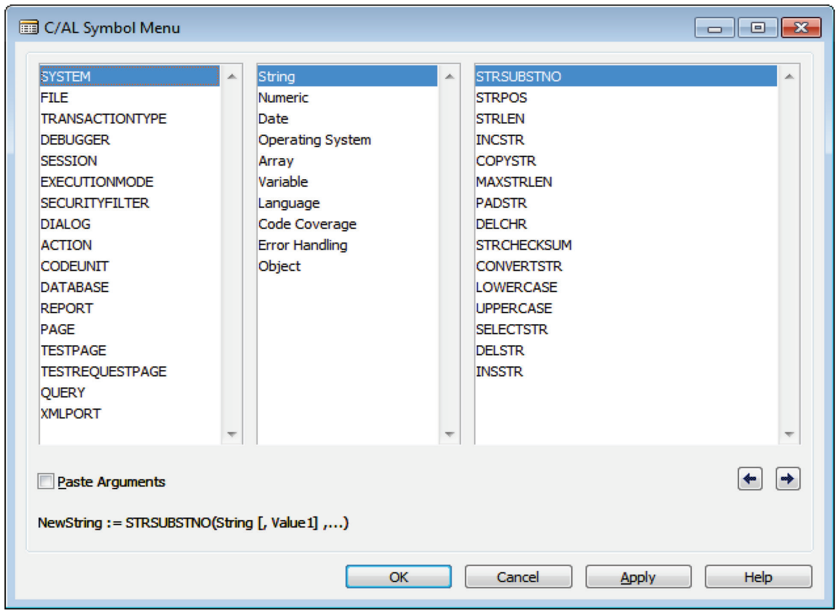
Review Built-in Functions

Use the **C/AL Symbol Menu** to review available variables and functions in the current scope of the trigger. It also shows the built-in functions that are available to you, together with their syntax.

The C/AL Symbol Menu

The following steps show how to open the C/AL Symbol Menu.

1. In the **Object Designer**, click **Codeunit** to access the codeunit list, and then click **New**.
2. Select the first line under the **OnRun** trigger.
3. Click **View** > **C/AL Symbol Menu**.
4. Click through the elements in the left column, and view the functions that are displayed.



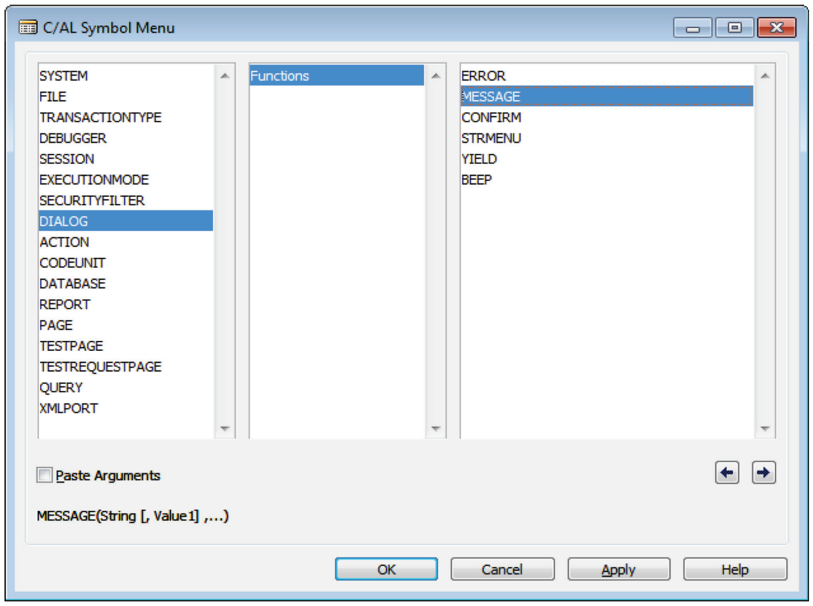
NewString := STRSUBSTNO(String [, Value1] ,...)

FIGURE 7.1: THE C/AL SYMBOL MENU

The MESSAGE Function

The following steps show how to review the MESSAGE function in the **C/AL Symbol Menu**.

1. In the left column of the **C/AL Symbol Menu**, select the **DIALOG** element to view the built-in functions.
2. In the right-side column, select the **MESSAGE** function.



MESSAGE(String [, Value1] ,...)

FIGURE 7.2: THE MESSAGE FUNCTION IN THE C/AL SYMBOL MENU

The syntax of the **MESSAGE** function is displayed at the bottom of the frame, above the command buttons in the **C/AL Symbol Menu** window.

The **MESSAGE** function has the following syntax:

Code Example

```
MESSAGE(String [, Value1] ...)
```

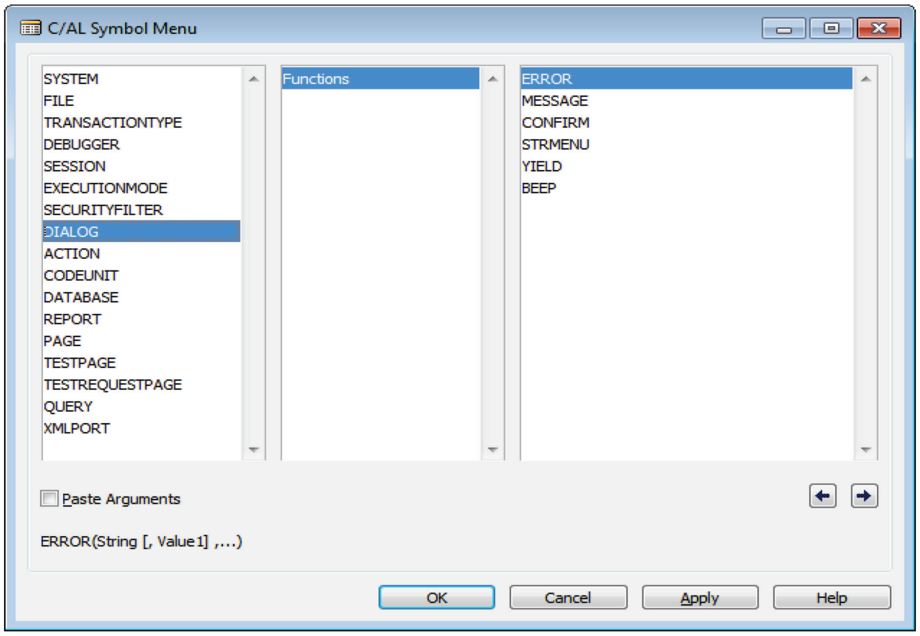
The syntax informs you of the following:

- There is no return value.
- There is one mandatory parameter (String).
- There are multiple optional parameters, (The square brackets indicate an optional parameter and the ellipsis (...) indicates multiples).

The ERROR Function

The following steps show how to review the **ERROR** function in the **C/AL Symbol Menu**.

1. In the left column of the **C/AL Symbol Menu**, select the **DIALOG** element.
2. In the right-side column, select the **ERROR** function.



ERROR(String [, Value1] ,...)

FIGURE 7.3: THE ERROR FUNCTION IN THE C/AL SYMBOL MENU

You can see the syntax of the **ERROR** function at the bottom of the **C/AL Symbol Menu**.

The **ERROR** function has the following syntax:

Code Example

```
ERROR(String [, Value1] ...)
```

The syntax of the **ERROR** function is identical to that of the **MESSAGE** function, except for the function identifier.

When an **ERROR** function is called in a statement, the processing stops with an error condition. The message is displayed in a similar manner to the **MESSAGE** function.

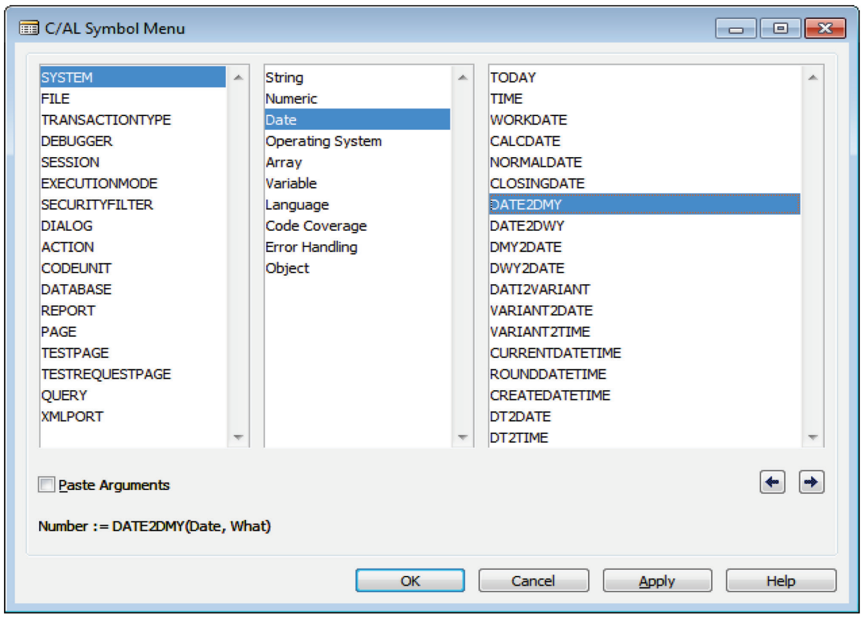
The DATE2DMY Function

The following steps show how to review the DATE2DMY function in the **C/AL Symbol Menu**.

1. In the left column of the **C/AL Symbol Menu**, select the **SYSTEM** element.

   The middle column lists function groupings, such as string functions, numeric (mathematical) functions, and so on. Depending on the selection in the middle column, the right-side column shows different functions.

2. In the middle column, select the **Date** group, and in the right-side column, select the **DATE2DMY** function.



Number := DATE2DMY(Date, What)

FIGURE 7.4: THE DATE2DMY FUNCTION IN THE C/AL SYMBOL MENU

The DATE2DMY function has the following syntax:

Code Example

```
Number := DATE2DMY(Date, What)
```

The syntax informs you of the following:

- The function has a return value that is a number, and is indicated by the "Number :=" right before the function identifier.
- The first parameter (Date) is an expression of type Date.
- The second parameter (What) is described in more detail in the online Help.

3. Press **F1**.

The **Microsoft Dynamics NAV Help** window opens and displays information for the DATE2DMY function. A complete description of the function role and values that it returns is shown. The What parameter is an integer expression that can take one of the following three values:

- If it is 1, this function returns the day of the month.
- If it is 2, this function returns the month (from 1 to 12).
- If it is 3, this function returns the year (the full 4 digits).

4. Close the codeunit without saving it.

Demonstration: Use the DATE2DMY Function

The following demonstration shows how to use the DATE2DMY function.

Demonstration Steps

1. Create a new codeunit, and save it as codeunit **90003**, **My Codeunit 4**.
   a. In **Object Designer**, click **Codeunit**.
   b. Click **New**.
   c. In the **C/AL Editor** window, on the **File** menu, click **Save**.
   d. In the **Save As** dialog box, in the **ID** field, enter "90003", then in the **Name** field, enter "My Codeunit 4", and then click **OK**.

2. Define variables.
   a. Click **View** > **C/AL Globals**.
   b. Define the following variables:

Name	DataType	Length
When Was It	Date	
Description	Text	30

3. Type the code in the **OnRun** trigger that displays a message with the name of the current month.
    a. Type the following code in the **OnRun** trigger:

Code Example

```
"When Was It" := TODAY;
CASE DATE2DMY("When Was It",2) OF
  1:Description := 'January';
  2:Description := 'February';
  3:Description := 'March';
  4:Description := 'April';
  5:Description := 'May';
  6:Description := 'June';
  7:Description := 'July';
  8:Description := 'August';
  9:Description := 'September';
  10:Description := 'October';
  11:Description := 'November';
  12:Description := 'December';
END;
MESSAGE('%1 is in %2',"When Was It",Description);
```

4. Compile, save, and then close the codeunit.
    a. Click **File** > **Save**.
    b. In the **Save** dialog box, make sure that the **Compiled** check box is selected.
    c. Click **OK**.
    d. Close the **C/AL Editor** window.

5. Run the codeunit and examine the result.
    a. In **Object Designer**, select codeunit **90003**.
    b. Click **Run**.

Data Access Functions

Accessing data is the most common task in a Microsoft Dynamics NAV 2013 application. In C/AL, Record is a complex data type that represents a single record in a table. There are several member functions on the Record data type that enable you to access rows in a table.

You can access rows either by iterating through a series of rows, or by directly accessing a specific row by referring to its primary key values.

Not all data access functions are equally important. This lesson presents the following functions:

- GET
- FIND
- FINDFIRST
- FINDLAST
- FINDSET
- NEXT

***Note:** All code examples in this lesson imply a WITH block on a variable of type Record.*

GET

The **GET** function retrieves one record, based on the value of the primary key. For example, if the **No.** field is the primary key of the **Customer** table, use the **GET** function as follows:

Code Example

```
GET(Customer,'30000');
```

The result is that the record of customer 30000 is retrieved.

The **GET** function produces a run-time error if it fails. The return value is not inspected by the code.

The following example shows how to inspect the return value of **GET** to avoid the run-time error.

Inspecting the return value of GET

```
IF GET(Customer,'4711') THEN

   .... // do some processing

 ELSE

   .... // do some error processing
```

The **GET** function searches for records, without regard to current filters, and it does not change any filters. It always searches among all records in a table.

FIND

The differences between the **GET** function and the **FIND** function are as follows:

- The **FIND** function respects (and is limited by) the current setting of filters.
- The **FIND** function can be instructed to look for records where the key value is equal to, larger than, or smaller than the current key values in a record.
- The **FIND** function can find the first or the last record, given the sorting order that is defined by the current key.

Use these features in a variety of ways. When you develop applications under a relational database management system, you frequently have one-to-many relationships between tables. An example is the relationships between the **Item** table that records items, and the **Sales Line** table that records the detail lines from sales orders. A record in the **Sales Line** table can only be related to one item. But each item can be related to any number of sales line records. An item record in the **Item** table must not be deleted while there are still open sales orders that include the item. The following code sample can be put in the **OnDelete** trigger of the **Item** table, and shows how to check for this condition:

Code Example

```
SalesOrderLine.SETCURRENTKEY("Document Type","No.");
SalesOrderLine.SETRANGE("Document Type",SalesOrderLine."Document
Type"::Order);
SalesOrderLine.SETRANGE(Type,SalesOrderLine.Type::Item);
SalesOrderLine.SETRANGE("No.","No.");
IF SalesOrderLine.FIND('-') THEN
  ERROR('You cannot delete because there are one or more outstanding sales
orders that include this item.');
```

***Best Practice:** You should avoid using the **FIND** function as much as you can. It is best to use **FINDFIRST**, **FINDLAST**, or **FINDSET**, depending on what you want to achieve.*

FIND, FINDFIRST, FINDLAST, FINDSET

To find the first record in a table, depending on the current key and filter, use the **FINDFIRST** function. You should use the **FINDFIRST** function instead of FIND('-') when only the first record is needed.

To find the last record in a table, depending on the current key and filter, use the **FINDLAST** function. You should use the **FINDLAST** function instead of FIND('+') when only the last record is needed.

To find a set of records in a table that correspond to a specified set of filters, use the **FINDSET** function. You should use the **FINDSET** function when you want to iterate through a record set, in combination with a REPEAT...UNTIL loop. The **FINDSET** function only lets you iterate from the first record to the last. To iterate from the last record to the first, you must use FIND('+') instead.

NEXT

The NEXT function is frequently used with **FIND** and **FINDSET** to step through records of a table. The following code sample shows how to use the **NEXT** function:

Code Example

```
FINDSET;
REPEAT
  // process record
UNTIL NEXT = 0;
```

Use the **FINDSET** function to locate the first record of the table. Afterward, use the **NEXT** function to step through every record, until there are no more (then, **NEXT** returns zero).

Sorting and Filtering Functions

When you access data from code, you seldom access all the rows in a table. You usually set the key to guarantee a specific ordering of the rows. You set filters to limit the record set to only those records that apply to the operation that you want to be completed.

Similar to functions such as **GET** or **FINDSET**, Record data type has several functions that manage key selection, sorting, and filtering.

This lesson presents the following functions:

- SETCURRENTKEY
- SETRANGE
- SETFILTER
- GETRANGEMIN
- GETRANGEMAX

SETCURRENTKEY

Use the **SETCURRENTKEY** function to select a key for a record. This sets the sorting order that is used for the associated table. It has the following syntax:

Code Example

```
[Ok :=] Record.SETCURRENTKEY(Field1, [Field2],...)
```

Following are several characteristics of the **SETCURRENTKEY** function:

- Inactive fields are ignored.
- When you search for a key, C/SIDE selects the first key that begins with the specified field(s).

For example, even if you specify only one field as a parameter when the code calls **SETCURRENTKEY**, the key that actually is selected may consist of more fields.

If no keys are found that include the specified field or fields, a run-time error occurs unless the Boolean return value of **SETCURRENTKEY** is handled in the code. If you handle the return value, you must decide what the program must do if the function returns *FALSE*. Without the run-time error, the program continues to run even though no matching key is found.

SETRANGE

The **SETRANGE** function sets a simple filter, such as a single range or a single value, on a field.

SETRANGE has the following syntax:

SETRANGE syntax

```
Record.SETRANGE(Field [,FromValue] [,ToValue])
```

If you specify both *FromValue* and *ToValue* parameters, the filter is a range filter, and includes all the records with the value in the specified field between the specified *FromValue* and *ToValue*. If you only specify *FromValue* parameter, the filter is a single-value filter, and includes all records which have exactly the value specified in *FromValue* in the specified field.

The **SETRANGE** function removes any previous filters on the specified field. If you omit both the *FromValue* and *ToValue* parameters, the function removes any filter that might already be set on the field.

The following code example shows how to use the SETRANGE function:

SETRANGE example

```
Customer.SETRANGE("No.", '10000', '30000');
```

This limits the Customer table by selecting only those records where the **No.** field has a value between 10000 and 30000.

SETFILTER

The **SETFILTER** function sets a filter in a more general way than **SETRANGE**. It allows you to specify filter expressions that include placeholders, operators, and wildcard characters.

The **SETFILTER** function has the following syntax:

Code Example

```
Record.SETFILTER(Field, String, [Value],...)
```

The *Field* parameter is the field on which you want to set a filter. The *String* parameter is a filter expression that can contain wildcard characters, operators, and placeholders, such as %1, %2, and so on to indicate locations where the system inserts values given as the *Value* parameters.

The following code example shows how to use the **SETFILTER** function:

SETFILTER example

```
Customer.SETFILTER("No.", '>10000 & <> 20000');
```

This code selects those records where the value in **No.** field is larger than 10000 and not equal to 20000.

The following code example achieves the same result as the previous one, but through placeholder syntax:

SETFILTER example with parameters

```
Customer.SETFILTER("No.",'>%1&<>%2',Value1, Value2);
```

GETRANGEMIN and GETRANGEMAX

The **GETRANGEMIN** and **GETRANGEMAX** functions retrieve the minimum or maximum value of the filter that is currently in effect on a field.

The **GETRANGEMIN** and **GETRANGEMAX** functions have the following syntax:

GETRANGEMIN and GETRANGEMAX syntax

```
Value := Record.GETRANGEMIN(Field)

Value := Record.GETRANGEMAX(Field)
```

The **GETRANGEMIN** and **GETRANGEMAX** functions cause a run-time error if the filter currently in effect is not a range. For example, if a filter is set as follows:

```
Customer.SETFILTER("No.",'10000|20000|30000');
```

Then the following code fails, because the filter is not a range:

```
BottomValue := Customer.GETRANGEMIN("No.");
```

Data Manipulation Functions

Applications do not only read data, they also change data. C/AL includes a series of functions that let you insert rows, change, or delete rows in a table. Even though the syntax and the purpose of those functions are different, most of these functions return a Boolean value that indicates whether they have succeeded or not. If you do not handle the return value in your code, and the function call fails, a run-time error occurs. If you handle the return value, then you have to decide what the program must do if the function returns as FALSE.

The following functions are the most common manipulation functions:

- INSERT
- MODIFY
- MODIFYALL
- RENAME
- DELETE
- DELETEALL

INSERT

The **INSERT** function inserts a record in a table. **INSERT** returns a Boolean value that indicates whether the insert operation succeeded or failed.

The following code sample shows how to use the INSERT function:

INSERT example

```
Customer.INIT;
Customer."No." := '4711';
Customer.Name := 'John Doe';
Customer.INSERT;
```

This code inserts a new record, with **No.** and **Name** having the assigned values, whereas the other fields have their default values. Supposing that **No.** is the primary key of the **Customer** table. The record is inserted in the **Customer** table unless there already is a record in the table with the same primary key. In that case, because the return value is not handled, a run-time error occurs.

MODIFY

The **MODIFY** function changes an existing record. **MODIFY** returns a Boolean value that indicates whether the modify operation succeeded or failed.

The following code sample shows how to use the **MODIFY** function:

MODIFY Example

```
Customer.GET('4711');
Customer.Name := 'Richard Roe';
Customer.MODIFY;
```

The code changes the name of customer 4711 to Richard Roe.

MODIFYALL

The **MODIFYALL** function performs a bulk update of records. It also sets the value of the specified field to the same specified value for all records. It respects the current filters. This means that you can perform the update on a specified set of records in a table.

The **MODIFYALL** function returns no value, nor does it cause an error if the set of records to be changed is empty.

The following code sample shows how to use the **MODIFYALL** function:

Code Example

```
Customer.SETRANGE("Salesperson Code",'PS');
Customer.MODIFYALL("Salesperson Code",'JR');
```

The **SETRANGE** statement selects the records where the **Salesperson Code** is “PS”. The **MODIFYALL** function changes these records to have the **Salesperson Code** set to JR.

DELETE

Use the **DELETE** function to delete a record from the database. The record to be deleted must be specified by using the values in the primary key fields before the program calls the function. The **DELETE** function takes filters into consideration.

DELETE returns a Boolean value that indicates whether the delete operation succeeded or failed.

The following code sample shows how to use the **DELETE** function:

DELETE example

```
Customer."No." := '4711';
Customer.DELETE;
```

DELETEALL

Use the **DELETEALL** function to delete all records that are selected by the filter settings. If no filters are set, all records in the table are deleted.

The following code sample shows how to use the **DELETEALL** function to delete all records where the **Salesperson Code** is PS from the **Customer** table:

DELETEALL Example

```
Customer.SETRANGE("Salesperson Code", 'PS', 'PS');
Customer.DELETEALL;
```

Similar to **MODIFYALL**, the **DELETEALL** function does not return the Boolean parameter. It does not fail if no record was affected.

Working with Fields

When you access data, you frequently have to perform actions on fields. For example, you may want to modify a value in a field, check whether any or a specific value is present in the field, or throw an error related to a field value. C/AL includes several functions which let you manipulate the field values and fields themselves.

The following functions are most frequently used with fields:

- CALCFIELDS
- SETAUTOCALCFIELDS
- CALCSUMS
- FIELDERROR
- FIELDCAPTION
- INIT
- TESTFIELD
- VALIDATE

CALCFIELDS

Use the **CALCFIELDS** function to calculate the **FlowFields**. If page controls or report columns reference a **FlowField**, the **FlowField** is automatically calculated. However, when you access a record variable from the code, then you must explicitly calculate the value of a **FlowField** that you intend to use.

***Note:** Any **FlowField** in a **Rec** or **xRec** variable in tables and pages is automatically calculated when the record is retrieved. You never have to call **CALCFIELDS** for FlowFields in these two built-in variables.*

The following code example shows how to use the **SETRANGE** function to set a filter and the **CALCFIELDS** function to calculate the FlowFields:

Code Example

```
SETRANGE("Date Filter",0D,TODAY);
CALCFIELDS(Balance,"Balance Due");
```

The **CALCFIELDS** function calculates the **Balance** and **Balance Due** fields by considering filter setting and calculations that are defined as the **CalcFormula** properties of the **FlowFields**.

In addition to calculating the value of a **FlowField**, the **CALCFIELDS** function also retrieves the values of BLOB fields from the database. BLOB fields only will be retrieved from the database if they are used as source expressions on page controls or data item columns in reports. If you want to access a BLOB value from the C/AL code, you must first call **CALCFIELDS**.

SETAUTOCALCFIELDS

Specifies the **FlowFields** that are automatically calculated whenever a record is retrieved from the database. When you are writing code there seems to be no difference between **CALCFIELDS** and **SETAUTOCALCFIELDS**. The **SETAUTOCALCFIELDS** function can improve performance, because it reduces the number of round trips to the database. Unlike **CALCFIELDS**, which always results in a separate database call, **SETAUTOCALCFIELDS** retrieves the values for the **FlowFields** in the same SQL Server operation as the other fields in the record.

Consider the following example:

CALCFIELDS and SETAUTOCALCFIELDS comparison

```
// Using CALCFIELDS

IF Customer.FINDSET THEN

  REPEAT

    Customer.CALCFIELDS(Balance,"Net Change");

    ... // Do some additional processing.

  UNTIL Customer.NEXT=0;

// Using SETAUTOCALCFIELDS

Customer.SETAUTOCALCFIELDS(Balance,"Net Change");

IF Customer.FINDSET THEN

  REPEAT

    // Customer.Balance and Customer."Net Change" are auto calculated.

    ... //Do some additional processing.

  UNTIL Customer.NEXT=0;
```

In the first block, the **CALCFIELDS** function retrieves the values of **Balance** and **Net Change** fields after each row is read from the database as a separate database operation. The **FINDSET** operation first selects the rows from the database as a single operation. In each iteration through the set, another database operation occurs that retrieves the **FlowFields** values.

In the second block, **SETAUTOCALCFIELDS** specifies that **Balance** and **Net Change** fields should be retrieved together with any other fields in the database. The **FINDSET** operation then retrieves the rows, together with the specified **FlowField** values in a single database operation.

CALCSUMS

The **CALCSUMS** function calculates the total of a column in a table. Similar to the **CALCFIELDS** function, the **CALCSUMS** function considers current filter settings when it performs the calculation.

The following code example shows how to select an appropriate key, set the filters, and perform the calculation:

CALCSUMS example

```
SETCURRENTKEY("Customer No.");
SETRANGE("Customer No.",'10000','50000');
SETRANGE(Date,0D,TODAY);
CALCSUMS(Amount);
```

For **CALCSUMS** to work properly, you do not have to specify the field as a **SumIndexField** in a table key. If you call **CALCSUMS** on a field that is not configured as a **SumIndexField**, then all rows are read and summed automatically.

Note: *Maintaining an index for a* ***SumIndexField*** *improves the performance of* ***CALCSUMS****, but may decrease the performance of insert, modify, and delete operations.*

FIELDERROR

The **FIELDERROR** function stops the execution of the code causing a run-time error. It also creates an error message for a field. It includes the field caption in the error message. This makes it very versatile, and guarantees consistent end-user experience across the application.

The following code sample shows how to use the **FIELDERROR** function:

FIELDERROR example

```
IF Item."Unit Price" < 10 THEN

  Item.FIELDERROR("Unit Price",'must be greater than 10');
```

This causes an appropriate message that shows, depending on whether **Class** currently is empty or has a value.

You can add custom text if the default text does not suit the application. The following code sample shows how to use the **FIELDERROR** function with a custom text:

FIELDCAPTION

The **FIELDCAPTION** function returns the caption of a field. By using the **FIELDCAPTION** function, you make sure that messages that you show to users still accurately reflect the field caption even if the field caption is changed later. You can use the **FIELDCAPTION** function together with the **FIELDERROR** function.

The following code sample shows how to call FIELDCAPTION together with FIELDERROR:

FIELDCAPTION example

```
FIELDERROR(Quantity,'must not be less than ' + FIELDNAME("Quantity Shipped"));
```

INIT

The **INIT** function initializes a record by assigning the default values for all non-primary key fields in the table. If a default value for a field is defined by the **InitValue** property, this value is used for the initialization; otherwise, the default value of each data type is used.

The INIT function does not initialize the fields of the primary key.

The following example shows how to use the **INIT** function:

INIT example

```
Customer.INIT;
```

TESTFIELD

The **TESTFIELD** function tests if the value of a field matches the specified value. If the values are not the same, the test fails, an error message is displayed, and a run-time error occurs. If no test value is specified, then the function tests if there is any value in the field.

Note: *Zero is treated as no value for numeric data types.* FALSE *is treated as no value for Boolean. Date (0D) and time (0T) values are treated as no value.*

The following code sample causes a run-time error:

TESTFIELD Example

```
Code := 'DK'
TESTFIELD(Code,'ZX');
```

VALIDATE

Use the **VALIDATE** function to call the OnValidate trigger of a field. It accepts an optional parameter that you can use to first set the value in the field to the specified value, and then to call the OnValidate trigger.

The following code sample shows how to use the **VALIDATE** function to call the OnValidate trigger of the **Total Amount** field:

VALIDATE example

```
VALIDATE("Total Amount");
```

The OnValidate trigger frequently checks the business rules and makes sure that no business rules are violated by trying to store an invalid value into a field. For example, the OnValidate trigger on the **Item No.** field of the **Item Journal Line** table makes sure that the item that is selected by the user is not blocked.

Additionally, the OnValidate trigger typically sets values to other fields in the same record, based on a value that is specified in the field being validated. For example, the OnValidate trigger on the **No.** field of the **Sales Line** table populates many other fields in the **Sales Line** table, such as **Description,** or **Unit of Measure Code**.

Note: *OnValidate trigger typically checks changes other values.*

User Interaction Functions

During code execution, the code frequently requires the user to provide certain inputs. For example, the application asks the user to confirm an invoice posting operation, or provide a selection of options before posting a sales order. Additionally, after an operation is complete, you may want to inform the user about the results of an operation. For example, after posting a journal, the application informs the user that the journal was posted successfully.

The following functions give users feedback and an opportunity to input information into the application:

- MESSAGE
- CONFIRM
- STRMENU
- ERROR

MESSAGE

The **MESSAGE** function displays a text string in a message window.

The **MESSAGE** function has the following syntax:

MESSAGE syntax

```
MESSAGE (String [, Value1, ...])
```

When a message statement in the C/AL code executes, the message is not immediately displayed. Instead, it is displayed after the C/AL code has finished executing, or after the C/AL code pauses to wait for user interaction.

The message window remains open until the user clicks **OK**.

Use the following characters for special purposes in the String parameter:

- The backslash character (\) starts a new line.
- The % symbol is a variable placeholder.

***Note:** When you use the % symbol, it automatically formats the variable, as if you called the **FORMAT** function to convert a variable into string.*

The following code example shows how to use the **MESSAGE** function:

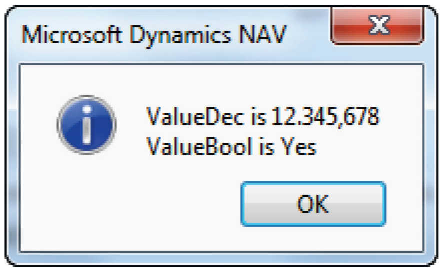
MESSAGE example

```
ValueDec := 12345.678;

ValueBool := TRUE;

MESSAGE('ValueDec is %1\ValueBool is %2',ValueDec,ValueBool);
```

When this code runs, the following result is displayed:



ValueDec is 12.345,678
ValueBool is Yes

FIGURE 7.5: THE MESSAGE DIALOG BOX

CONFIRM

The **CONFIRM** function creates a dialog box that prompts the user for a yes or no answer. The dialog box is centered on the screen.

The **CONFIRM** function has the following syntax:

CONFIRM syntax

Ok := Dialog.CONFIRM(*String* [, *Default*] [, *Value1*] ,...)

Like the **MESSAGE** function, **CONFIRM** also displays a message to the user. However, **CONFIRM** requires the user to answer a question by clicking **Yes** or **No** and returns a Boolean value (TRUE or FALSE), that corresponds to the user's selection. The **CONFIRM** function runs modally, and the application waits for the user's response before continuing the code execution.

The **CONFIRM** function frequently is used to confirm that the user wants to continue with a process. For example, Microsoft Dynamics NAV 2013 uses **CONFIRM** functions before posting records. The user is given an opportunity to stop the posting process or to continue.

The Default parameter specifies the default button in the message window. If it is FALSE, the **No** button is set as the active button. If the user only presses **ENTER**, the function returns FALSE. If the Default parameter is TRUE, the **Yes** button is set as the default button instead, and pressing **ENTER** returns TRUE.

Note: *You can also use the % placeholders with CONFIRM. However, you must remember to include the Default Boolean parameter before listing the variables that replace the placeholders.*

The following code example shows how to use the CONFIRM function:

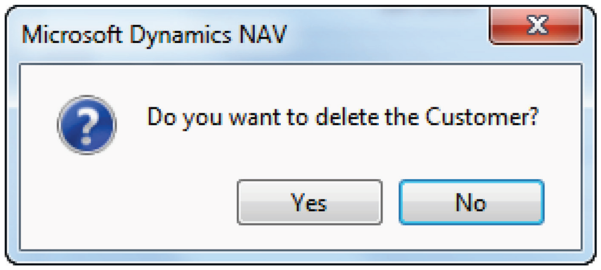
CONFIRM example

```
IF NOT CONFIRM('Do you want to delete the
%1?',FALSE,Customer.TABLECAPTION) THEN

  EXIT;

Customer.DELETE;
```

When this code runs, the result is as follows:



Do you want to delete the Customer?

FIGURE 7.6: THE CONFIRM DIALOG BOX

The CONFIRM function responses are limited to Yes and No. If other responses are needed, use the STRMENU function.

STRMENU

The **STRMENU** function creates a menu window that displays a series of options from a comma-delimited string.

The **STRMENU** function has the following syntax:

STRMENU syntax

```
OptionNumber := STRMENU(OptionString [, DefaultNumber] [, Instruction])
```

STRMENU returns an integer value that represents the user's selection. The first choice is 1, the second choice is 2, and so on. If the user clicks **Cancel** or presses ESC, then **STRMENU** returns 0.

If the *DefaultNumber* parameter is not set, the first option in automatically selected, with a value of one.

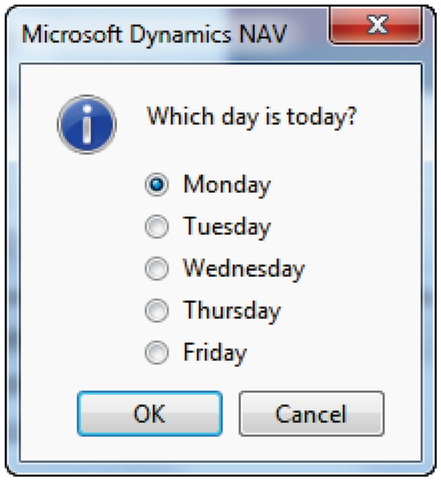
The following code sample shows how to use the **STRMENU** function:

STRMENU example

```
Days := 'Monday,Tuesday,Wednesday,Thursday,Friday';

Selection := STRMENU(Days,1,'Which day is today?');
```

When this code runs, the result is as follows:



Which day is today?

- [✓ Monday]
- Tuesday
- Wednesday
- Thursday
- Friday

FIGURE 7.7: THE STRMENU DIALOG BOX

ERROR

The **ERROR** function displays an error message and ends the execution of C/AL code.

The **ERROR** function has the following syntax:

ERROR syntax

```
ERROR(String [, Value1, ...])
```

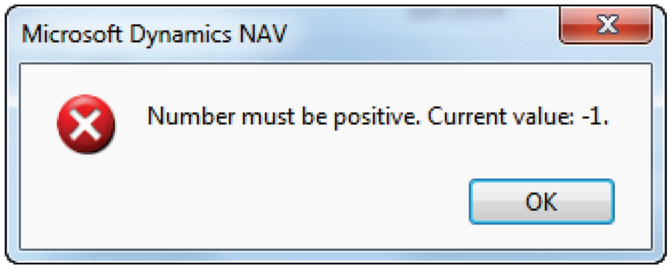
The **ERROR** function raises an error condition and leaves the current process, and cancels the entire process, not just the function. If the code is in a database transaction, the transaction stops and all uncommitted data roll back. The **ERROR** function displays an error message to the user. This message informs the user that additional processing is not allowed. The *String* parameter specifies this error message.

The following example demonstrates the **ERROR** function:

ERROR example

```
Number := -1;
IF Number <= 0 THEN BEGIN
  ERROR('Number must be positive. Current value: %1.', Number);
  MESSAGE('This Message is never displayed');
END;
```

When this code runs, the result is as follows:



Number must be positive. Current value: -1.

FIGURE 7.8: THE ERROR DIALOG BOX

Other Common C/AL Functions

In addition to data access and manipulation, C/AL language simplifies many other programming tasks. You frequently must manipulate strings, numbers, dates or times, or access some system functions, such as retrieving the current date or time, or looking up the current company name.

This lesson presents the most common functions in the following groups:

- String functions
- Date functions
- Number functions
- Array functions
- Stream functions
- System functions
- Other functions

Note: *There are additional function groups, and most of these groups contain more functions than presented in this lesson. To perform tasks in Microsoft Dynamics NAV Development Environment you only have to know the most common functions. If you want to know more, you can find more information about these and other functions in Microsoft Dynamics NAV Developer and IT Pro Help.*

String Functions

Manipulating strings is one of the more common programming tasks. You frequently must do any of the following tasks:

- Find an index of a substring within another string.
- Check the length of a string.
- Take a substring from a string.

There are several built-in C/AL functions that let you manipulate strings.

The following table lists the most common string manipulation functions, and explains their usage.

Function	Syntax	Remarks
STRPOS	Position := STRPOS(String,SubString)	Searches for the first occurrence of a substring in a string. It returns the position of the first character of the substring within the string. If the substring is not found within the string, the function returns a zero. As with all string functions, the **STRPOS** function is case-sensitive.
COPYSTR	NewString := COPYSTR(String, Position [, Length])	Copies a substring of any length from a specific position in a string to a new string. If you omit the Length parameter, the result includes all characters from Position to the end of the string.
STRLEN	Length := STRLEN(String)	Returns an integer that is the length of the string in the parameter.
MAXSTRLEN	MaxLength := MAXSTLEN(String)	Returns the maximum defined length of a string variable. You define the length when you declare a variable, or create a field in the table.

Function	Syntax	Remarks
LOWERCASE	NewString := LOWERCASE(String)	Converts all letters in a string to lowercase.
UPPERCASE	NewString := UPPERCASE(String)	Converts all letters in a string to uppercase.
INCSTR	NewString := INCSTR(String)	Increases a positive number or decreases a negative number inside a string by one. If *String* contains more than one number, then only the last number is changed. For example, 'SINV-13-3901' is changed to 'SINV-13-3902'.
SELECTSTR	NewString := SELECTSTR(Number, CommaString)	Retrieves a substring from a comma-separated string. The substrings are numbered starting with one.

***Note:** You can call all these functions on variables and fields both of type Text and Code.*

Date Functions

In a business application, such as Microsoft Dynamics NAV 2013, manipulating dates is a common programming task. The C/AL language provides several useful functions that modify or otherwise operate on date values.

The following table summarizes the most common date functions:

Function	Syntax	Remarks
DATE2DMY	Number := DATE2DMY(Date, What)	Retrieves the day, month, or year of a given date. The *What* parameter specifies which part of the date that you want to retrieve. It accepts the following values: • 1 for day • 2 for month • 3 for year

Function	Syntax	Remarks
DATE2DWY	Number := DATE2DWY(Date, What)	Retrieves the day of the week, week number, or year of a given date. The *What* parameter specifies which part of the date that you want to retrieve. It accepts the following values: • 1 for the day of the week (1 for Monday, 2 for Tuesday, and so on) • 2 for the week number in a year • 3 for year
CALCDATE	NewDate := CALCDATE(DateExpression [, Date])	Calculates a new date that is based on a date expression and a reference date. If you omit the optional *Date* parameter, the calculation is based on current system date. *DateExpression* can be of text, code, or DateFormula data type. For example, to calculate the first day of the next month, use the following code: **CALCDATE example** `NewDate := CALCDATE('<CM+1D>')` **Note:** *To learn more about date formulas, contact the Microsoft Dynamics NAV Developer and IT Pro Help.*

Function	Syntax	Remarks
NORMALDATE	NormalDate := NORMALDATE(Date)	Gets the regular date (instead of the closing date) for the specified date. You typically call the **NORMALDATE** function from posting routines to make sure that the user does not enter a closing date where a posting date is required. For example, a posting date on a sales invoice must not be a closing date.
CLOSINGDATE	ClosingDate := CLOSINGDATE(Date)	Gets the closing date for the specified date. A closing date is a time following the given date but before the next regular date. Closing dates are sorted immediately after the corresponding regular date, but before the next regular date.

Numeric Functions

There are several numeric functions that simplify certain operations with numbers, such as rounding.

The following table lists the numeric functions in C/AL:

Function	Syntax	Remarks
ROUND		Rounds the value of a numeric variable to a specified precision and in a specified direction. The *Direction* parameter can take the following values: • '=' rounds up or down to the nearest value. Values of 5 or more are rounded up. Values less than 5 are rounded down. • '>' rounds up. • '<' rounds down. If you omit the optional *Direction* parameter, then the

Function	Syntax	Remarks
		value is rounded to the nearest value. If you omit the *Precision* parameter, then the following steps determine the precision: 1. The function **ReadRounding** in Codeunit 1, **Application Management**, is called. **ReadRounding** returns a decimal value that is the precision. By default, the **ReadRounding** function returns the **Amount Rounding Precision** field from the **General Ledger Setup** table. 2. If you have customized Codeunit 1 and it does not implement the **ReadRounding** function, then the **No. of digits after decimal** in the **Regional and Language Options** on the current computer is used to specify the precision. If the **No. of digits after decimal** does not specify a valid value, then the precision is specified as two digits after the decimal.
ABS	NewNumber := ABS(Number)	Calculates the absolute value of a number (Decimal, Integer or BigInteger). **ABS** always returns a positive numeric value or zero.
POWER	NewNumber := POWER(Number, Power)	Raises a number to a power. For example, you can use this function to square the number 2 to obtain the result of 4.

Function	Syntax	Remarks
RANDOM	Number := RANDOM(MaxNumber)	Returns a pseudo-random number, between 1 and the value that is specified in the *MaxNumber* parameter. The number is pseudo-random, because it is always selected from the same sequence of numbers. To randomize the sequence, call the **RANDOMIZE** function.
RANDOMIZE	RANDOMIZE([Seed])	Generates a sequence of random numbers from which the **RANDOM** function selects random numbers. If you use the same number as the *Seed* parameter, the same set of numbers is generated. If you omit the Seed parameter, the total number of milliseconds since midnight is used. This guarantees unpredictable results.

Array Functions

In C/AL, you can declare an array of any data type. To help you manipulate the data in the array, C/AL provides several functions.

The following table lists the array functions:

Function	Syntax	Remarks
ARRAYLEN	Length := ARRAYLEN(Array[, Dimension])	Returns the total number of elements in an array or the number of elements in a specific dimension. If you omit the optional *Dimension* parameter, then the return value represents the total number of elements in the whole array.

Function	Syntax	Remarks
COMPRESSARRAY	[Count =:] COMPRESSARRAY(StringArray)	Moves all non-empty strings (text) in an array to the beginning of the array. The resulting *StringArray* has the same number of elements as the input array, but empty entries and entries that contain only blanks are displayed at the end of the array. The **COMPRESSARRAY** function only supports one-dimensional arrays. You cannot use this function together with multidimensional arrays.
COPYARRAY	COPYARRAY(NewArray, Array, Position[, Length])	Copies one or more elements from an array to a new array. The *Position* parameter specifies the position of the first array element to copy, whereas the optional *Length* parameter specifies the number of array elements to copy. If you omit the *Length* parameter, **COPYARRAY** copies all array elements from *Position* to the last element. The **COPYARRAY** function only supports one-dimensional arrays. You can call it repeatedly to copy more dimensions. ***Note:*** *You cannot copy arrays of complex data types.*

Stream Functions

Streaming is an important concept in programming languages. In C/AL it lets you access binary data that is stored in objects such as files or BLOB fields, without consideration of the actual representation, capabilities, or limitations of the object in which the data is actually stored. In addition to accessing data in files or BLOBs, C/AL lets you stream data directly from NAV and into NAV by using XMLports, and to exchange streams with .NET Framework objects through DotNet variables.

C/AL supports the following two data types for handling streams:

- **InStream** to read data from an object, such as a file or a BLOB.
- **OutStream** to write data to an object, such as a file or a BLOB.

The following table lists the C/AL functions that you use to manipulate streams:

Function	Syntax	Remarks

CREATEINSTREAM	File.CreateInStream(Stream)
	Blob.CreateInStream(Stream)
		Creates an InStream object for data reading from a file or a BLOB.

CREATEOUTSTREAM	File.CreateOutStream(Stream)
	Blob.CreateOutStream(Stream)
		Creates an OutStream object for data writing to a file or a BLOB.

READ	[Read :=] InStream.Read(Variable[, Length])	Reads a specified number of bytes from an InStream object. Data is read in binary format. Variables can be any data type. However, the data and the length must match the storage capacity of the variable. If the *Length* differs from the size of *Variable*, a run-time error occurs.

Function	Syntax	Remarks
READTEXT	[Read :=] InStream.ReadText(Text[, Length])	Reads text from an InStream object into a string variable. *Text* must be of a string data type, and the length must be shorter than, or equal to the maximum length of the *Text* variable. If you try to read more than the maximum length of the *Text* variable, a run-time error occurs.
WRITE	[Written :=] OutStream.Write(Variable[, Length])	Writes a specified number of bytes from the *Variable* to the stream. Data is written in binary format. For variables other than text, code, and binary, if you specify length that differs from the size of the variable, a run-time error occurs.
WRITETEXT	[Written :=] OutStream.WriteText([Text[, Length]])	Writes text to an OutStream object. If you omit the *Text* parameter, then an Enter and a line feed character are written.
EOS	IsEOS := InStream.EOS	Indicates whether an input stream has reached End of Stream (EOS). ***Note:*** *Reading past the end of stream results in a run-time error.*
COPYSTREAM	[Ok :=] COPYSTREAM(OutStream, InStream)	Copies all the data from an InStream to an OutStream.

Note: *Modules 9 XMLports, and 11 .NET Interoperability cover streaming in more detail.*

System Functions

From C/AL code, you frequently must access information about the operating system, the environment, or the current context of Microsoft Dynamics NAV 2013. C/AL provides several system functions that provide such information. System functions do not require any parameters because they return information that is stored in the system.

The following table shows the most important system functions:

Function	Remarks
USERID	Gets the ID of the currently logged-on user.
COMPANYNAME	Gets the current company name.
TODAY	Gets the current operating system date. You can only use the **TODAY** function to retrieve the current date from the operating system. You cannot use it to set the date in the operating system. ***Note:*** *Make sure not to use the* ***TODAY*** *function for setting defaults for dates that are relevant in financial transactions. Use* ***WORKDATE*** *instead.*
TIME	Gets the current operating system time.
WORKDATE	Gets and sets the work date for the current session. Every user can set the work date for their session. ***Best Practice:*** *Even though it is possible, the best practice is not to change the work date from code. You should let users control their work dates, and the code should not interfere with users' choice.*

Function	Remarks
TEMPORARYPATH	Gets the path of the directory where the temporary file for Microsoft Dynamics NAV is stored. If you intend to create any temporary files, then you should consider storing them in the directory that is returned by this function.
GUIALLOWED	Checks whether the C/AL code can display any information on the screen. If the code is running under Web services, or in a NAV Application Server (NAS) session, then **GUIALLOWED** returns FALSE. If the code is running under the RoleTailored client, then **GUIALLOWED** returns TRUE.
GLOBALLANGUAGE	Gets and sets the current global language setting. You can use this function to check the current language of the user session, or to change the language to any of the supported languages. The global language is represented by an integer value that corresponds to standard Windows language ID.

Other Functions

There are some other useful C/AL functions which help you manipulate the variables, convert between string and other simple data types, or manipulate the flow of code execution.

The following table shows these functions:

Function	Syntax	Remarks
EXIT	EXIT(Value)	Leaves the current function or trigger immediately. If there is a parent function that calls this current function or trigger, the *Value* parameter of the **EXIT** function is returned to the calling function. The **EXIT** function causes no error condition nor does it roll back any data.

Function	Syntax	Remarks
CLEAR	CLEAR(Variable)	Clears the value of a single variable. For simple data types, CLEAR resets the value to the default value for the variable data type. For complex data types, it resets the variable to its previous state before it first was accessed by the code. For example, for the record variables, **CLEAR** clears all the filters that were set, resets all the fields to their initial value (including the primary key fields), and resets the current key to the primary key.
CLEARALL	CLEARALL	Clears all user-defined variables in scope of the trigger from which **CLEARALL** is called. CLEARALL does not clear the Rec and xRec variables. CLEARALL works by calling the **CLEAR** function repeatedly on each variable.
EVALUATE	[Ok :=] EVALUATE(Variable, String[, Number])	Converts the *String* parameter into the actual data type of the *Variable* parameter, and stores the converted value in *Variable*. It returns a Boolean value that indicates whether the conversion was successful. If you do not inspect the return value and EVALUATE fails, then a run-time error occurs.
FORMAT	String := FORMAT(Value[, Length][, FormatStr/FormatNumber])	Formats a value into a string. You may specify the desired length of the resulting string. For numeric, date, and time values, you can also select the format into which the value is translated.

Create Custom Functions

Creating custom functions makes code more efficient. Creating functions includes adding, removing, or editing code once in a function that is used several times in an application. The change then is applied every time the function is called.

Reasons to Create Custom Functions

Reasons to create custom functions include the following:

- **To organize the program.** A function is to code what headings are to a well-organized document. Functions make programs easier to follow for you and other developers who may have to modify the code later.
- **To make code self-explanatory.** When you extract a segment of code to a function, and assign a meaningful name to the function, your code becomes easier to understand.
- **To simplify the development process.** When you design a program, you can break a complex problem into multiple smaller tasks. Each of these tasks can become a function in the program, and you can build the whole program from these smaller tasks. If a function is too complex, you can break it apart into smaller tasks and create a new function for each task.
- **To make code reusable and reduce work.** If the same or very similar things are performed in two separate parts of a program, consider creating a function to do that task. Instead of writing the same code in two or three locations, write it in one location and call it from other locations.
- **To reduce the possibility of errors.** You can thoroughly test a single function by calling it with all possible or likely combinations of parameters. When you are searching for errors, you can reduce the search. If an error is found in a function, you can fix it in one place and it is automatically fixed in all locations that called that function. If you had to fix the same code in different locations, you might forget one or add another bug.
- **To make modifications easier.** If you have to modify the way a program works and there is a similar code in many locations, the modifications must be made to each of those locations. If you put the common code in a function, you change one location and every other location that uses that function uses the updated code. This reduces work and reduces the possibility of introducing more errors.

- **To isolate data.** When a function performs a task, it can have its own local data that cannot be tampered with by other functions in the same object. By using its own local data, a function does not tamper with data that is owned by other functions. If many functions use the same global variables, this data can become corrupted.
- **To reduce the size of the objects.** Although it is a minor consideration, this is worthwhile when you try to locate something in the code.

There are many reasons to create functions. When you design code for an object, the first task is to determine major tasks and then create, or define, a function for each one. When you organize similar tasks in different locations, consider adding another function to handle such tasks.

Another reason to create functions that is specific to C/SIDE is that functions are one of the main ways to communicate between objects. Objects cannot make variables available to other objects. However, they can expose functions which access those variables.

Local Functions, Variables and the EXIT Statement

Some functions and variables have a limited scope. They can be used only in the location where they are defined.

Local Function

A local function can only be called from inside the object in which it is defined. When you define new functions, by default they are not local. They can be called both from inside the object in which they are defined, and from other objects.

Local Variable

A *local variable* is a variable with the scope limited to a single trigger. This means that code in a particular function can access a local variable and use it like any other variable. However, the code in other functions in the object cannot access it. If the name of a local variable is referred to outside the function in which it is declared, a compile-time error occurs. The formal parameters of a function are also treated as local variables in that function.

Note: *To define local variables, click* ***View*** *>* ***C/AL Locals****.*

The EXIT Statement

Use the **EXIT** statement to stop the execution of a trigger or a function. In functions, you use the EXIT statement to return a value from the function.

When you use a function call in an expression, the function must return a value. When you write a function that has a return value, use the **EXIT** statement to signal to the system to return a value.

The EXIT statement has the following syntax:

EXIT syntax

```
EXIT(<expression>)
```

For example, create a function named **Square** to square a value.

The following expression results in the Answer variable being assigned the value 29.

EXIT example

```
Answer := 4 + Square(5);
```

If the formal parameter is named *Param*, you can write the **Square** function trigger code as follows:

Square Function

```
EXIT(Param * Param);
```

Note: *Instead of using **EXIT**, you can give a name to the return parameter. If you do this, then you can assign the return value to the return parameter. The function exists with the value that was last assigned to the return parameter.*

Lab 7.1: Create Custom Functions

Scenario

Viktor is a business systems developer for Cronus International Ltd. He is learning how to develop customizations for Microsoft Dynamics NAV 2013, and how to program in C/AL to customize solutions for his customers.

He wants to customize the **Customer Card** page by adding an action that adjusts a customer's credit limit based on the customer's previous sales history, or to reset the limit to zero if there were no transactions.

The requirements state the following:

- A customer's credit limit may not exceed 50% of total sales revenue for the customer in the past twelve months.
- Customer's credit limit must always be rounded to the nearest 10,000.
- If the credit limit is rounded during the process, the application must send notification to the user.
- If the new credit limit does not differ from the old one, the application must send notification to the user.
- If the customer has no sales history over the past twelve months, the credit limit must be reset to zero.
- The function that sets the credit limit must be available to other objects.
- The **Customer Card** page must include an action that calls the function.
- If the function is called from the page action, and credit limit would be increased, the function must ask the user for confirmation before actually updating the credit limit.

Objectives

The objectives are:

- Create local and global functions.
- Create a page action that calls a function.
- Pass parameters by reference and by value.
- Use ROUND, SETRANGE, CALCFIELDS, CALCDATE, WORKDATE, CONFIRM, VALIDATE, FIELDCAPTION and MODIFY functions.

Exercise 1: Create Functions

Exercise Scenario

Viktor starts by analyzing the requirements. One requirement states that the function that updates the credit limit must be available to other objects. This indicates a global function. The best practice in Microsoft Dynamics NAV is never to store business logic in page objects. Therefore, this function must be located in another object that is available by the page and any other object. Because the function handles business logic about how to change a specific entity (**Customer**), the best location to put the global function that updates the credit limit for a customer is table 18, **Customer**.

However, another requirement states that if the user calls this function from a page action, the application first should ask for confirmation. This indicates two additional functions:

- One global function on table **Customer** that calculates the new credit limit.
- One local function on page **Customer Card** that should call other functions and ask for confirmation.

Therefore, Viktor creates three functions to perform the following tasks:

- Calculate the new credit limit.
- Update the credit limit.
- Handle the confirmation message.

Task 1: Add Global Functions to Customer Table

High Level Steps

1. Add a global function to table 18, **Customer**, and name it **UpdateCreditLimit**.
2. Modify the **UpdateCreditLimit** function so that it receives a parameter named *NewCreditLimit*, of type Decimal, by reference.
3. Add another global function to table 18, **Customer**, and name it **CalculateCreditLimit**.
4. Modify the **CalculateCreditLimit** function to return a value of type **Decimal**.
5. Compile and save the table 18, **Customer**.

Detailed Steps

1. Add a global function to table 18, **Customer**, and name it **UpdateCreditLimit**.
   a. In **Object Designer**, click **Table**.
   b. Select table 18, **Customer**, and then click **Design**.

   c. Click **View > C/AL Globals** to open the **C/AL Globals** window.
   d. On the **Functions** tab, in the first blank line, enter "UpdateCreditLimit".
   e. Do not close the **C/AL Globals** window.

2. Modify the **UpdateCreditLimit** function so that it receives a parameter named *NewCreditLimit*, of type Decimal, by reference.
   a. Make sure that the **UpdateCreditLimit** function is selected on the **Functions** tab of the **C/AL Globals** window.
   b. Click **Locals** to open the **C/AL Locals** window.
   c. On the **Parameters** tab, in the first line of the **Name** column, enter "NewCreditLimit".
   d. In the **DataType** column, enter **Decimal**, and then select the **Var** check box.
   e. Close the **C/AL Locals** window.
   f. Do not close the **C/AL Globals** window.

3. Add another global function to table 18, **Customer**, and name it **CalculateCreditLimit**.
   a. In the **C/AL Globals** window, on the **Functions** tab, in the first blank line enter "CalculateCreditLimit".
   b. Do not close the **C/AL Globals** window.

4. Modify the **CalculateCreditLimit** function to return a value of type **Decimal**.
   a. Make sure that **CalculateCreditLimit** is selected on the **Functions** tab of the **C/AL Globals** window.
   b. Click **Locals** to open the **C/AL Locals** window.
   c. On the **Return Value** tab, in the **Return Type** field, enter **Decimal**.
   d. Close the **C/AL Locals** window.
   e. Close the **C/AL Globals** window.

5. Compile and save the table 18, **Customer**.
   a. Click **File > Save**.
   b. In the **Save** dialog box, make sure that the **Compiled** check box is selected.
   c. Click **OK**.
   d. Close the **Table Designer** for table 18, **Customer**.

Task 2: Add a Local Function to Customer Card Page

High Level Steps

1. Add a local function to page 21, **Customer Card**, and name it **CallUpdateCreditLimit**.
2. Compile, save, and close page 21, **Customer Card**.

Detailed Steps

1. Add a local function to page 21, **Customer Card**, and name it **CallUpdateCreditLimit**.
   a. In **Object Designer**, click **Page**.
   b. Select page 21, **Customer Card** and then click **Design**.
   c. Click **View** > **C/AL Globals**.
   d. On the **Functions** tab, in the first empty line, enter "CallUpdateCreditLimit".
   e. Make sure that the **CallUpdateCreditLimit** function is selected, and click **View > Properties**.
   f. In the **Local** property, enter **Yes**.
   g. Close the **Properties** window.

2. Compile, save, and close page 21, **Customer Card**.
   a. Click **File > Save**.
   b. In the **Save** dialog box, make sure that the **Compiled** check box is selected.
   c. Click **OK**.
   d. Close the Page Designer for page 21, **Customer Card**.

Results

Customized objects:

Table 18, **Customer**

Page 21, **Customer Card**

Exercise 2: Add Action to Page

Exercise Scenario

Now that all required functions are in place, Viktor is ready to customize the **Customer Card** page to include an action that calculates the credit limit. He adds a new action to the **ActionItems** action container and to the **Functions** action group. He also promotes the action to the Process group of the **Home** tab, and assigns an image to it.

Task 1: Add an Action to Customer Card Page

High Level Steps

1. Add an action to the **Function** action group of **ActionItems** action container. Set its caption to **Update Credit Limit** and its name to **UpdateCreditLimit**.
2. Make the **Update Credit Limit** action call the **CallUpdateCreditLimit** function.
3. Set properties on the **Update Credit Limit** action to promote it to the **Process** category as a big action.
4. Set a property on the **Update Credit Limit** action to show the **CalculateCost** image.
5. Compile, save, and close page 21, **Customer Card**.

Detailed Steps

1. Add an action to the **Function** action group of **ActionItems** action container. Set its caption to **Update Credit Limit** and its name to **UpdateCreditLimit**.
   a. In **Object Designer**, click **Page**.
   b. Select page 21, **Customer Card**, and then click **Design**.
   c. Click **View > Page Actions**.
   d. In the **Action Designer** window, locate and select the first action in the **Functions** action group of the **ActionItems** action container. It should be **ApplyTemplates**.
   e. Click **Edit > New**.
   f. In the **Caption** column, enter "Update Credit Limit", and then in the **Name** column enter "UpdateCreditLimit".

2. Make the **Update Credit Limit** action call the **CallUpdateCreditLimit** function.
   a. In **Action Designer**, make sure that **Update Credit Limit** action is selected.
   b. Click **View > C/AL Code**.
   c. In the **UpdateCreditLimit - OnAction** trigger, type the following code:

```
CallUpdateCreditLimit;
```

   d. Close the **C/AL Editor** window.

3. Set properties on the **Update Credit Limit** action to promote it to the **Process** category as a big action.
   a. In the **Action Designer**, make sure that the **Update Credit Limit** action is selected.

   b. Click **View** > **Properties**.
   c. Set **Promoted** to **Yes**.
   d. Set **PromotedCategory** to **Process**.
   e. Set **PromotedIsBig** to **Yes**.
   f. Do not close the **Properties** window.

4. Set a property on the **Update Credit Limit** action to show the **CalculateCost** image.
   a. In the **Image** property, click the **Lookup** button to open the **Image List** window.
   b. In the Image **List** window, select the **CalculateCost** line.
   c. Click **OK**.
   d. Close the **Properties** window.
   e. Close the **Action Designer** window.

5. Compile, save, and close page 21, **Customer Card**.
   a. Click **File** > **Save**.
   b. In the **Save** dialog box, make sure that the **Compiled** check box is selected.
   c. Click **OK**.
   d. Close the **Page Designer** for page 21, **Customer Card**.

Results

Customized object:

Page 21, **Customer Card**

Exercise 3: Add Code to Functions

Exercise Scenario

Viktor adds code to functions, to complete the customization. He changes the **CalculateCreditLimit** function in table 18 (**Customer**) to perform the calculation according to the requirements. He then changes the **UpdateCreditLimit** function in table 18, **Customer**, to update the credit limit. Finally, Viktor adds code to the **CallUpdateCreditLimit**, to call the **CalculateCreditLimit** function, and ask for confirmation.

Task 1: Add Code to CalculateCreditLimit Function

High Level Steps

1. In table 18, **Customer**, in the **CalculateCreditLimit** function, declare a local variable named *Cust*, of type **Record** 18, **Customer**.
2. Add code to the **CalculateCreditLimit** function, that calculates the **Sales (LCY)** field for the past twelve months.

Detailed Steps

1. In table 18, **Customer**, in the **CalculateCreditLimit** function, declare a local variable named *Cust*, of type **Record** 18, **Customer**.
   a. In Object **Designer**, click **Table**.
   b. Select table 18, **Customer**, and then click **Design**.
   c. Click **View** > **C/AL Globals**.
   d. In the **C/AL Globals** window, on the **Functions** tab, select the **CalculateCreditLimit** function, and then click **Locals**.
   e. In the **C/AL Locals** window, on the **Variables** tab, in the **Name** column, enter "Cust" in the **DataType** column enter **Record**, and in the **SubType** column, enter **Customer**.
   f. Close the **C/AL Locals** window.

2. Add code to the **CalculateCreditLimit** function, that calculates the **Sales (LCY)** field for the past twelve months.
   a. In the **C/AL Globals** window, on the **Functions** tab, right-click the **CalculateCreditLimit** function.
   b. In the pop-up menu, click **Go To Definition**.
   c. In the **CalculateCreditLimit** function body, enter the following code:

```
Cust := Rec;

Cust.SETRANGE("Date Filter",CALCDATE('<-12M>',WORKDATE),WORKDATE);

Cust.CALCFIELDS("Sales (LCY)","Balance (LCY)");

EXIT(ROUND(Cust."Sales (LCY)" * 0.5));
```

   d. Close the **C/AL Editor**.

Task 2: Add Code to UpdateCreditLimit Function

High Level Steps

1. Add code to **UpdateCreditLimit** function, to round the *NewCreditLimit* formal parameter to the nearest 10,000, to validate the value of *NewCreditLimit* into the **Credit Limit (LCY)** field, and to modify the record.
2. Compile, save, and close the table 18, **Customer**.

Detailed Steps

1. Add code to **UpdateCreditLimit** function, to round the *NewCreditLimit* formal parameter to the nearest 10,000, to validate the value of *NewCreditLimit* into the **Credit Limit (LCY)** field, and to modify the record.
   a. On the **View** menu, click **C/AL Globals**.
   b. On the **Functions** tab, right-click the **UpdateCreditLimit** function.
   c. In the pop-up menu, click **Go To Definition**.
   d. In the body of the **UpdateCreditLimit** function, enter the following code:

```
NewCreditLimit := ROUND(NewCreditLimit,10000);

Rec.VALIDATE("Credit Limit (LCY)",NewCreditLimit);

Rec.MODIFY;
```

   e. Close the **C/AL Editor**.

2. Compile, save, and close the table 18, **Customer**.
   a. Click **File** > **Save**.
   b. In the **Save** dialog box, make sure that the **Compiled** check box is selected.
   c. Click **OK**.
   d. Close the **Table Designer** window for table 18, **Customer**.

Task 3: Add Code to CallUpdateCreditLlimit Function

High Level Steps

1. Add three global text constants to the page 21, **Customer Card**, with the text for:
   - Confirmation message
   - Information about the rounded credit limit
   - Information about an up-to-date credit limit.

2. Add two local decimal variables to the **CallUpdateCreditLimit** function, and call them *CreditLimitCalculated* and *CreditLimitActual*.
3. Add the code that calculates the new credit limit.
4. Add the code that checks if the credit limit is up to date. If it is, the code informs the user, and exits the function.
5. Add code that asks the user for confirmation of the credit limit update, and leaves the function if the user declines the confirmation.
6. Add code that updates the credit limit, and then informs the user if the rounding has occurred.
7. Compile, save, and close the page 21, **Customer Card**.

Detailed Steps

1. Add three global text constants to the page 21, **Customer Card**, with the text for:
   - Confirmation message.
   - Information about the rounded credit limit.
   - Information about an up-to-date credit limit.

   a. In **Object Designer**, click **Page**.
   b. Select page 21, **Customer Card**, and then click **Design**.
   c. Click **View** > **C/AL Globals**.
   d. On the **Text Constants** tab, enter the following three constants:

Name	ConstValue
Text90001	Are you sure that you want to set the %1 to %2?
Text90002	The credit limit was rounded to %1 to comply with company policies.
Text90003	The credit limit is up to date.

2. Add two local decimal variables to the **CallUpdateCreditLimit** function, and call them *CreditLimitCalculated* and *CreditLimitActual*.
   a. On the **Functions** tab, select the **CallUpdateCreditLimit** function, and then click **Locals**.
   b. In the **C/AL Locals** window, on the **Variables** tab, enter the following variables:

Name	DataType
CreditLimitCalculated	Decimal
CreditLimitActual	Decimal

3. Add the code that calculates the new credit limit.
   a. On the **Functions** tab, right-click the **CallUpdateCreditLimit** function.
   b. On the pop-up menu, click **Go To Definition**.
   c. In the body of the **CallUpdateCreditLimit** function, enter the following code:

```
CreditLimitCalculated := Rec.CalculateCreditLimit;
```

4. Add the code that checks if the credit limit is up to date. If it is, the code informs the user, and exits the function.
   a. In the body of the **CallUpdateCreditLimit** function, append the following code:

```
IF CreditLimitCalculated = Rec."Credit Limit (LCY)" THEN BEGIN

  MESSAGE(Text90003);

  EXIT;

END;
```

5. Add code that asks the user for confirmation of the credit limit update, and leaves the function if the user declines the confirmation.
   a. In the body of the **CallUpdateCreditLimit** function, append the following code:

```
IF GUIALLOWED AND NOT CONFIRM(

  Text90001,

  FALSE,

  FIELDCAPTION("Credit Limit (LCY)"),

  CreditLimitCalculated)

THEN

  EXIT;
```

6. Add code that updates the credit limit, and then informs the user if the rounding has occurred.
   a. In the body of the **CallUpdateCreditLimit** function, append the following code:

```
CreditLimitActual := CreditLimitCalculated;

Rec.UpdateCreditLimit(CreditLimitActual);

IF CreditLimitActual <> CreditLimitCalculated THEN

  MESSAGE(Text90002,CreditLimitActual);
```

The final version of the code in the **CallUpdateCreditLimit** function should be this:

CallUpdateCreditLimit Function

```
CreditLimitCalculated := Rec.CalculateCreditLimit;

IF CreditLimitCalculated = Rec."Credit Limit (LCY)" THEN BEGIN

  MESSAGE(Text90003);

  EXIT;

END;

IF GUIALLOWED AND NOT CONFIRM(

  Text90001,

  FALSE,

  FIELDCAPTION("Credit Limit (LCY)"),

  CreditLimitCalculated)

THEN

  EXIT;

CreditLimitActual := CreditLimitCalculated;

Rec.UpdateCreditLimit(CreditLimitActual);

IF CreditLimitActual <> CreditLimitCalculated THEN

  MESSAGE(Text90002,CreditLimitActual);
```

7. Compile, save, and close the page 21, **Customer Card**.
    a. Click **File** > **Save**.
    b. In the **Save** dialog box, make sure that the **Compiled** check box is selected.
    c. Click **OK**.
    d. Close the **Page Designer** for page 21, **Customer Card**.

Task 4: Test the Page

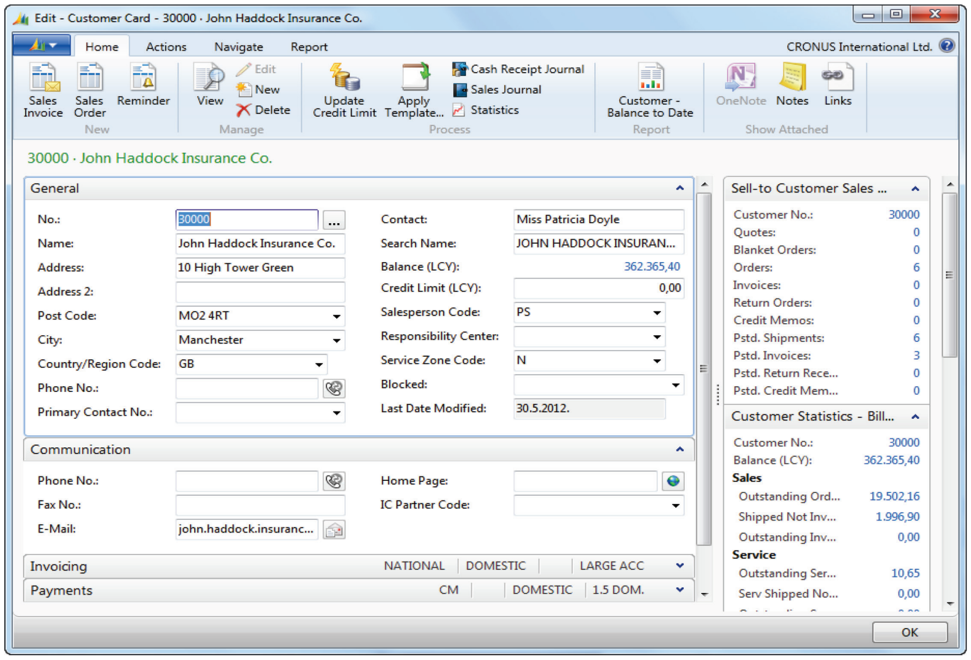
High Level Steps

1. In the client for Windows, open the **Customer Card** page for customer 30000, John Haddock Insurance Co.
2. Click the **Update Credit Limit** action to verify its functionality.

Detailed Steps

1. In the client for Windows, open the **Customer Card** page for customer 30000, John Haddock Insurance Co.
    a. Start the Microsoft Dynamics NAV client for Windows.
    b. In the **Search box**, enter “Customers” and press **ENTER**.
    c. Double-click the customer 30000, John Haddock Insurance Co.

The following figure shows the **Customer Card** page after customization, with the Update Credit Limit action promoted:



30000 · John Haddock Insurance Co.

General

No.:	30000	Contact:	Miss Patricia Doyle
Name:	John Haddock Insurance Co.	Search Name:	JOHN HADDOCK INSURAN...
Address:	10 High Tower Green	Balance (LCY):	362.365,40
Address 2:		Credit Limit (LCY):	0,00
Post Code:	MO2 4RT	Salesperson Code:	PS
City:	Manchester	Responsibility Center:	
Country/Region Code:	GB	Service Zone Code:	N
Phone No.:		Blocked:	
Primary Contact No.:		Last Date Modified:	30.5.2012.

Communication

Phone No.:		Home Page:	
Fax No.:		IC Partner Code:	
E-Mail:	john.haddock.insuranc...		

Invoicing NATIONAL | DOMESTIC | LARGE ACC

Payments CM | DOMESTIC | 1.5 DOM.

Sell-to Customer Sales ...

Customer No.:	30000
Quotes:	0
Blanket Orders:	0
Orders:	6
Invoices:	0
Return Orders:	0
Credit Memos:	0
Pstd. Shipments:	6
Pstd. Invoices:	3
Pstd. Return Rece...	0
Pstd. Credit Mem...	0

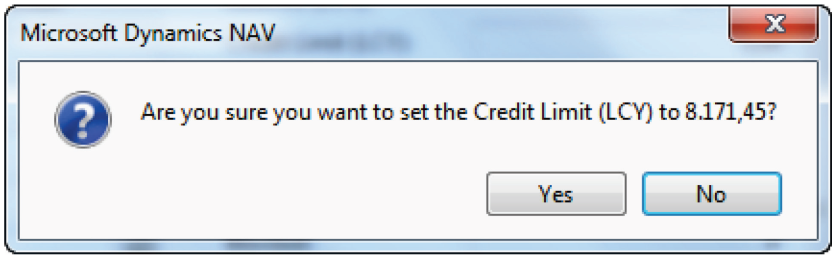
Customer Statistics - Bill...

Customer No.:	30000
Balance (LCY):	362.365,40
Sales	
Outstanding Ord...	19.502,16
Shipped Not Inv...	1.996,90
Outstanding Inv...	0,00
Service	
Outstanding Ser...	10,65
Serv Shipped No...	0,00

FIGURE 7.9: THE CUSTOMIZED CUSTOMER CARD PAGE

2. Click the **Update Credit Limit** action to verify its functionality.
   a. In the customer card, click the **Update Credit Limit** action.

The application asks for confirmation and displays the following dialog box:



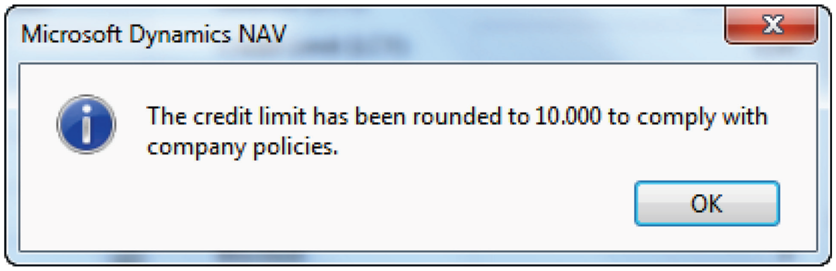
Microsoft Dynamics NAV

Are you sure you want to set the Credit Limit (LCY) to 8.171,45?

FIGURE 7.10: CREDIT LIMIT CONFIRMATION DIALOG

   b. Confirm the question in the dialog box.

The application informs the user that rounding has occurred:



Microsoft Dynamics NAV

The credit limit has been rounded to 10.000 to comply with company policies.

FIGURE 7.11: MESSAGE DIALOG

   c. Click **OK**.

Module Review

Module Review and Takeaways

C/SIDE provides many built-in functions that you can use in your code. These built-in functions are predefined with their syntax. C/SIDE also lets developers create custom functions to extend their application.

Understanding the concepts of functions, when to use built-in functions, and when to create custom functions helps developers efficiently develop custom solutions for Microsoft Dynamics NAV 2013.

Test Your Knowledge

Test your knowledge with the following questions.

1. How many local variables does the **CreateAccountingPeriodFilter** function in codeunit 358, DateFilter-Calc declare?

2. How many parameters does the **CreateAccountingDateFilter** function in codeunit 358, DateFilter-Calc have? How many of these are by reference, and how many are by value?

3. Which function call retrieves only the last record in the filter from the Customer table?

   ( ) Customer.FIND;

   ( ) Customer.FINDLAST;

   ( ) Customer.FIND('+');

   ( ) Customer.FIND('>');

   ( ) Customer.FINDSET;

4. Which function retrieves a record by its primary key values?

5. The MODIFY function returns FALSE or fails if no records were modified?

   ( ) True

   ( ) False

6. The MODIFYALL function returns FALSE or fails if no records were modified?

   ( ) True

   ( ) False

Test Your Knowledge Solutions

Module Review and Takeaways

1. How many local variables does the **CreateAccountingPeriodFilter** function in codeunit 358, DateFilter-Calc declare?

   MODEL ANSWER:

   None.

2. How many parameters does the **CreateAccountingDateFilter** function in codeunit 358, DateFilter-Calc have? How many of these are by reference, and how many are by value?

   MODEL ANSWER:

   5 parameters. 2 are by reference, 3 are by value.

3. Which function call retrieves only the last record in the filter from the Customer table?

   ( ) Customer.FIND;

   (√) Customer.FINDLAST;

   ( ) Customer.FIND('+');

   ( ) Customer.FIND('>');

   ( ) Customer.FINDSET;

4. Which function retrieves a record by its primary key values?

   MODEL ANSWER:

   GET

5. The MODIFY function returns FALSE or fails if no records were modified?

   (√) True

   ( ) False

6. The MODIFYALL function returns FALSE or fails if no records were modified?

   ( ) True

   (√) False