Extend standard SAP functionality of user exits through field symbols to meet custom business needs

by Owen McGivney

Owen McGivney
SAP HR/Payroll Consultant,
Bright Star HR Ltd.,
Richmond, London

Owen McGivney has worked on implementing and supporting SAP HR and payroll systems since 1998. While concentrating on UK payroll, he has also delivered Irish and multinational payroll solutions across a wide range of private and public sector clients. He has a special interest in combining ABAP programming with standard R/3 configuration to create effective solutions. You may reach him at owen@eridami.com.

SAP consultants and developers must regularly deliver solutions outside of standard SAP functionality to meet specialized business requirements. This leads them to search for suitable *user exits*, those predefined "hooks" in which the developer can insert customized code. All too often, however, they find a user exit that would just about solve a problem but cannot be used because it is missing one or more essential parameters — a field, structure, internal table, or other data element. This frequently happens because a user exit has not passed all the parameters that a developer needs to develop a solution, even though those parameters may actually be available in the program that calls the user exit. Lacking these parameters, the user exit becomes useless, sending the consultant in search of a completely new solution.

What this article shows, however, is that it is often possible to access those "missing" parameters with a few lines of code, rendering the user exit usable after all. This is a simple method of accessing the data in the main program from within the user exit by using field symbols. This approach can be a lifesaver for developers and support consultants, and yet is not widely known. (For an explanation of why these parameters may be missing, see the sidebar on page 55.)

The article demonstrates how you can apply this approach to developing solutions, taking user exits and features from the Human Resources (HR) module as examples. Although the examples are based on HR, the basic method will also be of interest to anyone working in other areas, since the same techniques can be applied to work in any other SAP R/3 module. This method should be useful for functional consultants and ABAP developers alike. Functional consultants, especially those who are able to identify user exits, will find that they can use this approach for giving detailed requirements to the developer.

This article begins by outlining a typical scenario in which you must investigate available HR user exits to design a custom enhancement. After

No portion of this publication may be reproduced without written consent.

a brief overview of methods for finding suitable exits, the discussion turns its focus on what to do if no suitable user exit can be found, and why this might be. If the main reason is that one or more key parameters have not been made available to the *user exit function module*, then this simple code method may solve the problem — provided some basic groundwork is first laid out.

This involves undertaking some basic investigations in the development, enhancement, and ABAP run-time environments using transactions SE38 (ABAP Editor), SE37 (ABAP Function Modules), SMOD (SAP Enhancement Management), CMOD (Enhancements), and the ABAP debugger. For those unfamiliar with the relevant transactions, a sequence of screenshots shows how to investigate the environment in which a user exit function module is situated. This serves two purposes. First, it gives you a greater understanding of how a function module exit is called from within a program and demonstrates the difference between global and local parameters — all of which will be very important later in the process. Second, it makes it possible for you to determine the exact parameters, program names, and other specific details that will be required later for the code; plus, by using the field and table display windows in the ABAP Debugger,[1] you can prove that these will work before the code is even written.

The next section will, I hope, prove especially useful. It offers sample code listings with detailed notes for three typical scenarios. These code listings have been formatted so that they can be used as templates; all you have to do is replace the text in blue with your own specific data and program references.

The method used in this article is powerful, but it must be used with care. For this reason, I carefully spell out the risks associated with the method and offer pointers about how to minimize potential problems. I also explain what factors to take into account when using this approach, and under what conditions it may be preferred to other, more usual, approaches.

[1] For information on the most recent ABAP Debugger, see the article "Introducing the next generation of ABAP debugging — the New ABAP Debugger" by Boris Gebhardt and Christoph Stöck (*SAP Professional Journal*, January/February 2006).

Prerequisites

The article is intended for, but certainly not limited to, anyone familiar with HR functionality and experienced at customizing by using the IMG. It also assumes familiarity with the concept of user exits and does not give comprehensive instructions on how to find or implement them. This information is already well documented in published articles, developer forums, and SAP online documentation. Some familiarity with the ABAP debugging environment and a basic understanding of variables and simple programming are required.

ABAP developers with a good knowledge of user exits (transactions CMOD and SMOD) and field symbols may want to skip the introductory sections and go straight to the code examples.

Basic method for investigating user exits and parameters

SAP provides user exits in the Implementation Guide (IMG), which you use to set up customized processes outside of standard SAP functionality to meet specialized business requirements. The documentation accompanying a user exit lists the parameters available for customization. (Alternatively, you can see the parameters by viewing the import and export tabs of the function module within SAP Enhancement Management through transaction SMOD.) But what can you do when a requisite parameter does not appear in the available user exit?

Dig deeper! To do this, you can employ a very basic method, which involves:

- Identifying a suitable user exit function module
- Analyzing the calling program

 ©2008 SAP Professional Journal. All rights reserved.

- Setting breakpoints
- Confirming the names of the fields, structures, and calling program

To begin the process, you need to identify a user exit function module.

Identifying a suitable user exit function module

Imagine company ABC is having difficulties integrating its payroll and HR processes on SAP. A recent merger and reorganization has left the payroll and HR

Note!

The terms *enhancement* and *user exit* are often used interchangeably to refer to the opportunities provided by SAP to enable clients to put in their own custom code to extend standard functionality. This can be confusing because enhancements can consist of multiple components, each of which may call a separate customer routine. In HR, most of these calls have been implemented as function modules. Therefore, technically speaking, the point in a standard piece of SAP code at which a customer routine is called should be referred to as a user exit function module. These are grouped together by functional relevance and assigned to enhancements. However, like most consultants I have met, I generally use the term "user exit" to cover both the higher level enhancement and the sublevel components that actually call the customer includes. I have used the term "user exit function module" to denote the latter whenever it seemed required for clarity.

Why are those parameters missing, anyway?

It's worth pausing to consider the reasons why a key parameter or internal data table may not be available within the user exit. SAP has defined each user exit with a given set of parameters, which it considers adequate for the intended purpose. There are several possible reasons why the parameters provided are insufficient for your purposes. The first thing to consider is that you may have misunderstood the purpose of the user exit under view and that a more appropriate user exit may exist elsewhere. It is also possible that SAP wants to forestall any changes being made to the field at the point at which the user exit function module is called because this would lead to problems later on with the program function. Make sure you have investigated all available options and followed the entire process from end to end so that you can identify, understand, and eliminate these possibilities.

There are also other reasons why an important parameter may not be available. This may happen when the system has been set up to use a standard data field in a non-standard way. For example, in many systems the fields on the Organizational Assignment infotype 0001 (such as contract, administrator group, and Org Key) have been used to store client-specific, high-level groupings that represent company structure, hierarchy, or employee characteristics. This is frequently the case when an HR system suffers from a poorly designed or unconventional enterprise or employee structure. It may also occur when an existing functional enterprise structure requires modification at a later date because of a change in the system scope or a major business reorganization, such as a merger.

Beyond this, it may simply be that the requirement at hand differs from what was envisaged by SAP when the user exit was developed.

No portion of this publication may be reproduced without written consent.

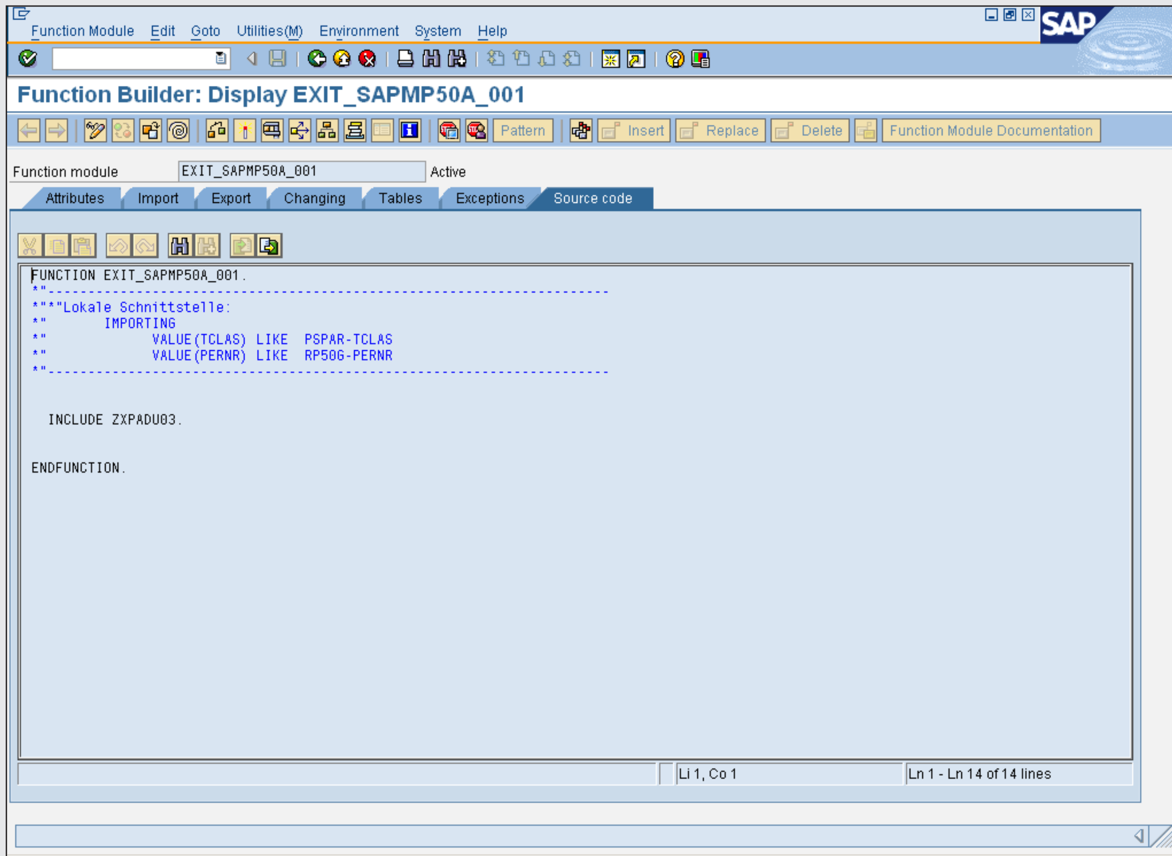

Figure 1 User exit EXIT_SAPMP50A_001

administration teams in widely different locations. The HR teams do not always remember the payroll cycle dates and, as a result, data changes are sometimes being made after payroll cut-off, causing headaches for the payroll department when it comes to balancing their figures. The payroll team has requested a solution that will output a warning message to remind the HR administrators not to make any changes until they exit payroll. The requirements for this warning are:

- It should be shown in the initial screen in Personnel Administration transaction PA30 (Maintain HR Master Data).
- It should be shown in Personnel Administration and not Recruitment.
- In particular, it should only appear for payroll-relevant employees (e.g., salaried employees) and not for non-payroll-relevant employees (e.g., contractors, temporary employees, etc.). This must be determined by employee group and/or payroll area.

Since there is no way of configuring this solution, you need to find a user exit with the requisite parameters for this development.

SAP Enhancements (user exits) are maintained in transaction SMOD, so let's carry out a search in this transaction to find a suitable user exit. By drilling down through the application components and concentrating on Personnel Administration, you can identify a potential user exit: EXIT_SAPMP50A_001

 ©2008 SAP Professional Journal. All rights reserved.

(Personnel Administration Menu: Function for Input of Personnel Number; see **Figure 1**). This exit calls up a customer function whenever a user enters a personnel number on the Personnel Administration initial screen in transaction PA30, at least partially meeting the first requirement. But a careful examination of this user exit function module shows that it only makes two parameters available to the user-accessible part of the code:

- The *transaction class* (tclas) can have one of two values: A = employees (i.e., normal employees in Personnel Administration) or B = applicants (if Recruitment has been implemented)
- The personnel number (pernr)

> ***Tip!***
>
> Transaction SMOD allows you to search for user exits by name, description, or application area (e.g., Personnel Management). SAP has flagged most user exits in the IMG in a node of the functional area to which they relate. Alternatively, a search in transaction SE37 with the string "EXIT_*" will display a list of all available user exits within the system. You can also identify other potentially useful user exits by searching the many lists, forums, and bulletin boards available on the Internet.

Since it passes parameters for the employee number and the transaction class, this user exit allows the routine to differentiate between employees in Recruitment and Personnel Administration. You can add logic to derive the payroll processing start and end dates. However, the requirement for this particular business need is to produce a response depending on the employee group, so that permanent and temporary employees can be processed differently. As it stands, this requirement cannot be met because there is no direct way of determining the employee type (i.e., the user exit function module gives us only the parameter for the employee number, not the employee type). Since at this point it is not possible to determine the employee group, you might well be inclined to discard the user exit as unsuitable and try to find another way of implementing the required processing. But such a decision would be premature because you may be able to derive the missing field if it is available in the main program, provided you carry out some basic investigation.

Analyzing the calling program

To investigate the user exit to determine if it is suitable, your next step is to find all of the programs that the user exit function module calls. This step has the dual benefits of discovering all the details that will be required in the code later and making it possible to prove that it will work before any code is actually written. In the process, you need to answer the following questions:

- What is the technical name of any program that calls the user exit?
- What are the technical names of any additional data fields or structures that might be required?
- Is the user exit used in more than one place?
- Does anything else happen to this data in the main program of which I should be aware?
- Does anything else happen, either before or after calling the routine, of which I should be aware?

> ***Note!***
>
> The remainder of this example concentrates on solving the problem of identifying the "unknown" employee group, employee subgroup, and payroll area from within the user exit function module. A full solution to the payroll warning message requirement above is not given since the code to d erive payroll dates and output warnings does not require any new or special programming techniques.

> ***Note!***
>
> The EXIT_SAPMP50A_001 calls a customer include named ZXPADU03. This means that the developer needs to create a section of code called ZXPADU03, which will then be included in the program flow. Note that this include has to be created, assigned to a project (transaction CMOD), and activated before it can be used. This is done in a standard way for all user exits, so I won't elaborate on this any further. If in doubt, consult an ABAP developer or systems administrator.

The first two questions have specific technical answers. The rest of the questions are more general, and their purpose is to ensure that a complete investigation of the program logic is carried out. This will help avoid wasting time and also mitigate potential program errors. (ABAP developers will already be able to determine the answers to these questions directly without the need for further instruction and can skip ahead to the code listings section.)

From the Function Builder screen (**Figure 1**) display the Where-Used List dialog by clicking any letter in EXIT_SAPMP50A_001 and clicking the Where-Used List button (or press Ctrl–Shift–F3).

In the dialog (shown in **Figure 2**), select Programs, Classes, and BSP Applications as the default options, and then click Execute**.** A list with all of the routines that call the user exit function module appears, as shown in **Figure 3**.

This user exit is called only by one program (MP50AF00). This means that you need to consider only one calling program. If more than one main routine calls the user exit function module, you must examine each calling program individually.

Drill down into this program by selecting the check box next to the program name and then clicking Display.

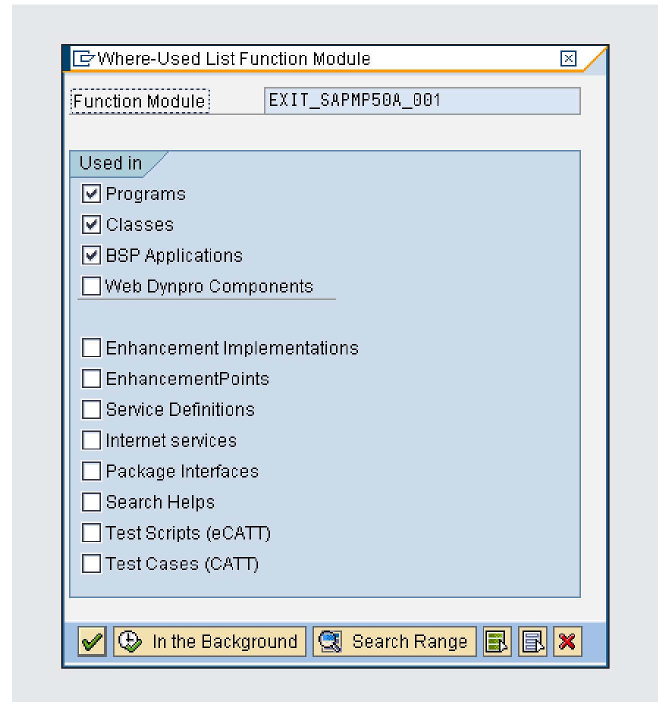

Figure 2 Where-Used List dialog for a user exit function module

> ***Tip!***
>
> If you already know the name of the calling program, you can find the point where the user exit function module is called by viewing the calling program in transaction SE38 and searching for the string `CALL CUSTOMER-FUNCTION`.

Figure 4 shows the point at which the main program calls the user exit. The line of code `CALL CUSTOMER-FUNCTION '001'` refers to EXIT_SAPMP50A_001, which in turn calls the user include ZXPADU03, but you can't see this name until you double-click the object. As expected, the only parameters exported to the user exit are the transaction class (tclas) and personnel number (pernr).

Examine the status of the available data fields, both before calling the user exit and within the user exit.

 www.SAPpro.com ©2008 SAP Professional Journal. All rights reserved.

Objects Edit Goto Utilities(M) Extras Environment System Help

Where-used Function Module EXIT_SAPMP50A_001 (1 Hits)

Complete List

Program	Short description
MP50AF00	SPECIAL_SUBFEATURE_CHECK

Figure 3 Where-Used Hit list for a user exit function module

Program Edit Goto Utilities(M) Environment System Help

ABAP Editor: Display Include MP50AF00

Pattern Insert Replace Delete Undo

Include MP50AF00 Active

```
    CALL CUSTOMER-FUNCTION '001'
         EXPORTING
              tclas   = pspar-tclas
              pernr   = rp50g-pernr.
    IF pspar-tclas EQ 'A'.                "QNUP40K034820
*     objtype = 'EMPLOYEET'.            "QNUP40K034820 (del) QICAHRK064434
      l_otype   = 'P'.                    "QICAHRK064434
    ELSE.                                 "QICAHRK064434
*     objtype = 'APPLICANT'.            "QNUP40K034820 (del) QICAHRK064434
      l_otype   = 'AP'.                   "QICAHRK064434
    ENDIF.                                "QNUP40K034820
*   CONCATENATE rp50g-pernr sy-datum "QNUP40K034820 (del) QICAHRK064434
*          INTO objkey.                 "QNUP40K034820 (del) QICAHRK064434
*---begin --- XFYAHRK062445 ------- deactivated ----------------------
**  CALL FUNCTION 'SWU_OBJECT_PUBLISH'                    "QNUP40K034820
**       EXPORTING                                        "QNUP40K034820
**            objtype           = objtype                 "QNUP40K034820
**            objkey            = objkey                  "QNUP40K034820
**             CREATOR           = ' '                    "QNUP40K034820
**          TABLES                                        "QNUP40K034820
**             CONTAINER         =                        "QNUP40K034820
**       EXCEPTIONS                                       "QNUP40K034820
**            objtype_not_found = 1                       "QNUP40K034820
**            OTHERS            = 2.                      "QNUP40K034820
*--- end ---- XFYAHRK062445 ------- deactivated ----------------------
    call function 'RH_OBJECT_PUBLISH'                     "QICAHRK064434
         exporting                                        "QICAHRK064434
              PLVAR                    = '  '             "QICAHRK064434
              otype                    = l_otype          "QICAHRK064434
              objid                    = rp50g-pernr      "QICAHRK064434
              SINGLE_OBJECT            = 'X'              "QICAHRK064434
         EXCEPTIONS                                       "QICAHRK064434
              INTERNAL_ERROR           = 1                "QICAHRK064434
              OTYPE_NOT_SUPPORTED      = 2                "QICAHRK064434
```

Li 288, Co 72 Ln 258 - Ln 292 of 2022 lines

Figure 4 Finding the point at which the user exit function module is called

No portion of this publication may be reproduced without written consent.

Setting breakpoints

You need to set a *breakpoint*, or debug point, at the line where the user exit is called so you can examine the status of all the parameters in which you are interested. I have set a breakpoint at the point at which the function module (above the line `CALL CUSTOMER-FUNCTION '001'`) is called using the breakpoint button in the ABAP Editor toolbar. At this point, you should also scroll up the ABAP Editor screen to read through the program logic from the start and read on further below to get an initial idea of what is going on around the user exit call.

For those unfamiliar with user exit function modules, it is worth clarifying that although there is a line calling a customer function, it doesn't do anything yet. You have to create the customer include, add your code, and activate it. Creating the include is a very straightforward task:

1. Double-click `CALL CUSTOMER-FUNCTION '001'`. Click Yes in the message box to create the object (see **Figure 5**).
2. Add a one-line program, as shown in **Figure 6**.
3. Save the include, add it to a transport, and activate the user exit.

Two break points have now been set up in the code: one created by using the breakpoint button in the ABAP Editor immediately before the user exit function module is called (which I mentioned earlier) and the second using the ABAP break-point command within the user exit function module itself. Now you can use the debugger to see exactly what

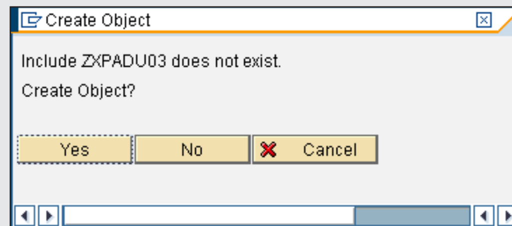

Figure 5 Creating a customer include object

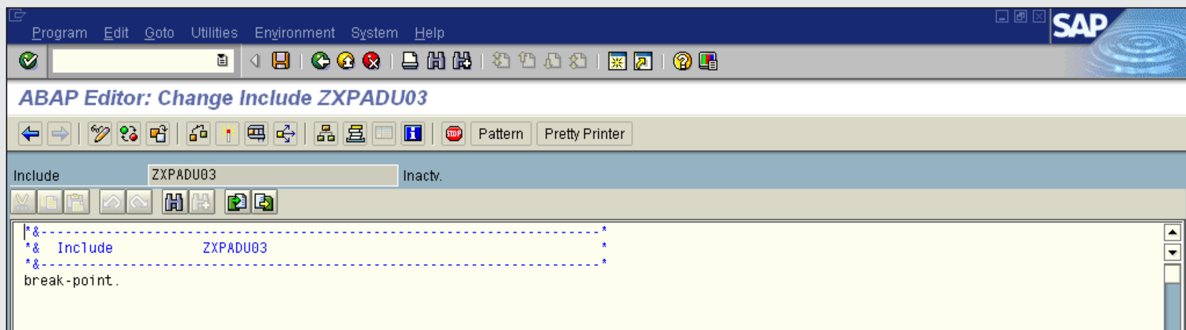

Figure 6 Example customer include with a breakpoint

 ©2008 SAP Professional Journal. All rights reserved.

happens to the program variables when the user exit is called. To do this, open a new session of Maintain HR Master Data in transaction PA30 and enter the *personnel number* of an active employee on your system. After pressing the Return key, the process-before-output (PBO) routines of the transaction are processed and the first breakpoint is eventually reached, as shown in **Figure 7**.

At this point, the program has run up to the debugger breakpoint, just before the user exit is called. The window at the top of the ABAP debugger shows that the main program name is SAPMP50A. The source code of window indicates that the main program has called include MP50AF00, which you saw in **Figure 3**. You need to write down the name of the main program because you will need it later on. *It is essential that the name of the calling program be correct for this approach to work.*

Four field names have been added to the field name window to view the variable values. At this point, the employee group (p0001-persg), employee subgroup (p0001-persk), transaction type (pspar-tclas), and personnel number (rp50g-pernr) are all available, as shown in the field windows below the code. **Figure 8** on the next page shows the fields and their corresponding values.

Using the ABAP Debugger, you can now investigate what happens when the next line of code

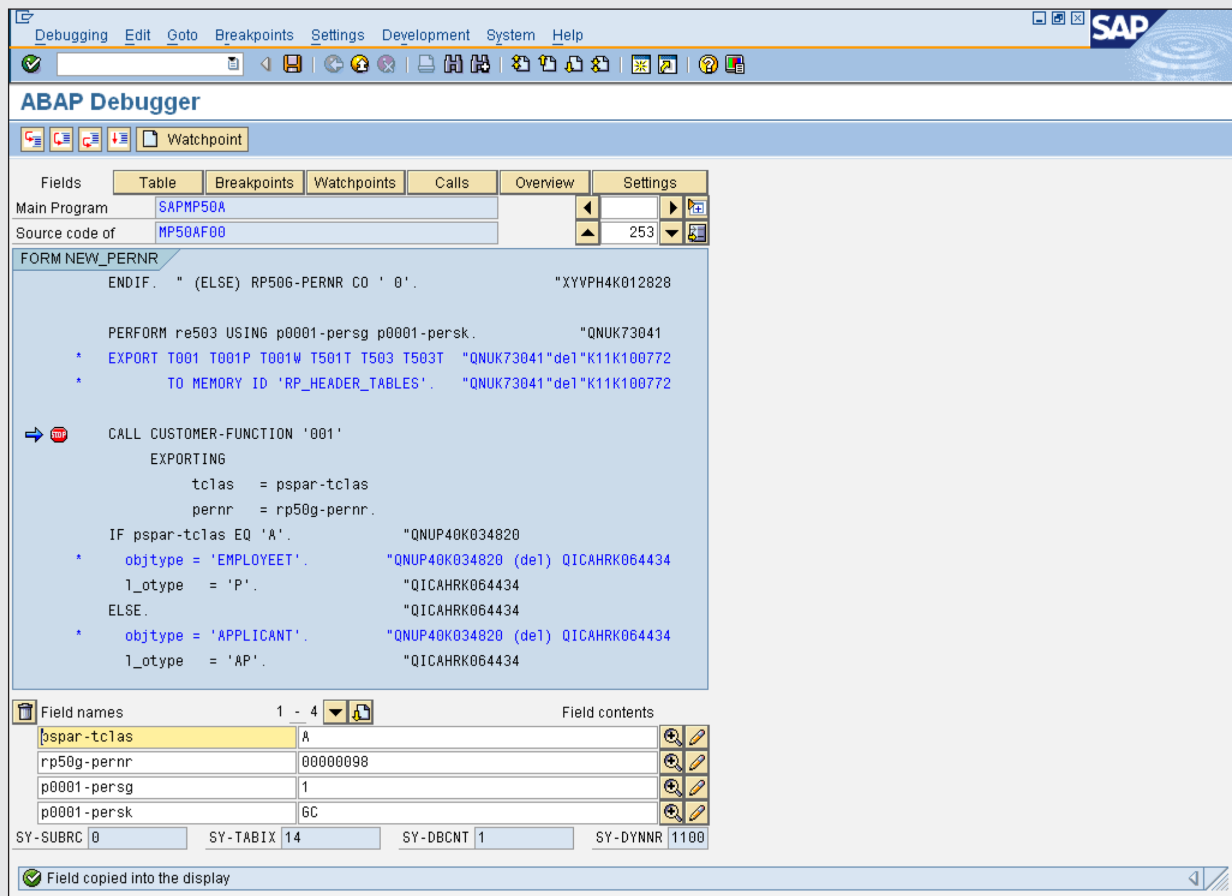

Figure 7 Debugger breakpoint before the user exit function module

No portion of this publication may be reproduced without written consent.

Field	Value	Comment
pspar-tclas	A	Transaction class
rp50g-pernr	00000098	Personnel number
p0001-persg	1	Employee group = 1 for active employees
p0001-persk	GC	Employee subgroup = GC for salaried staff

Figure 8 Field values before user exit is called

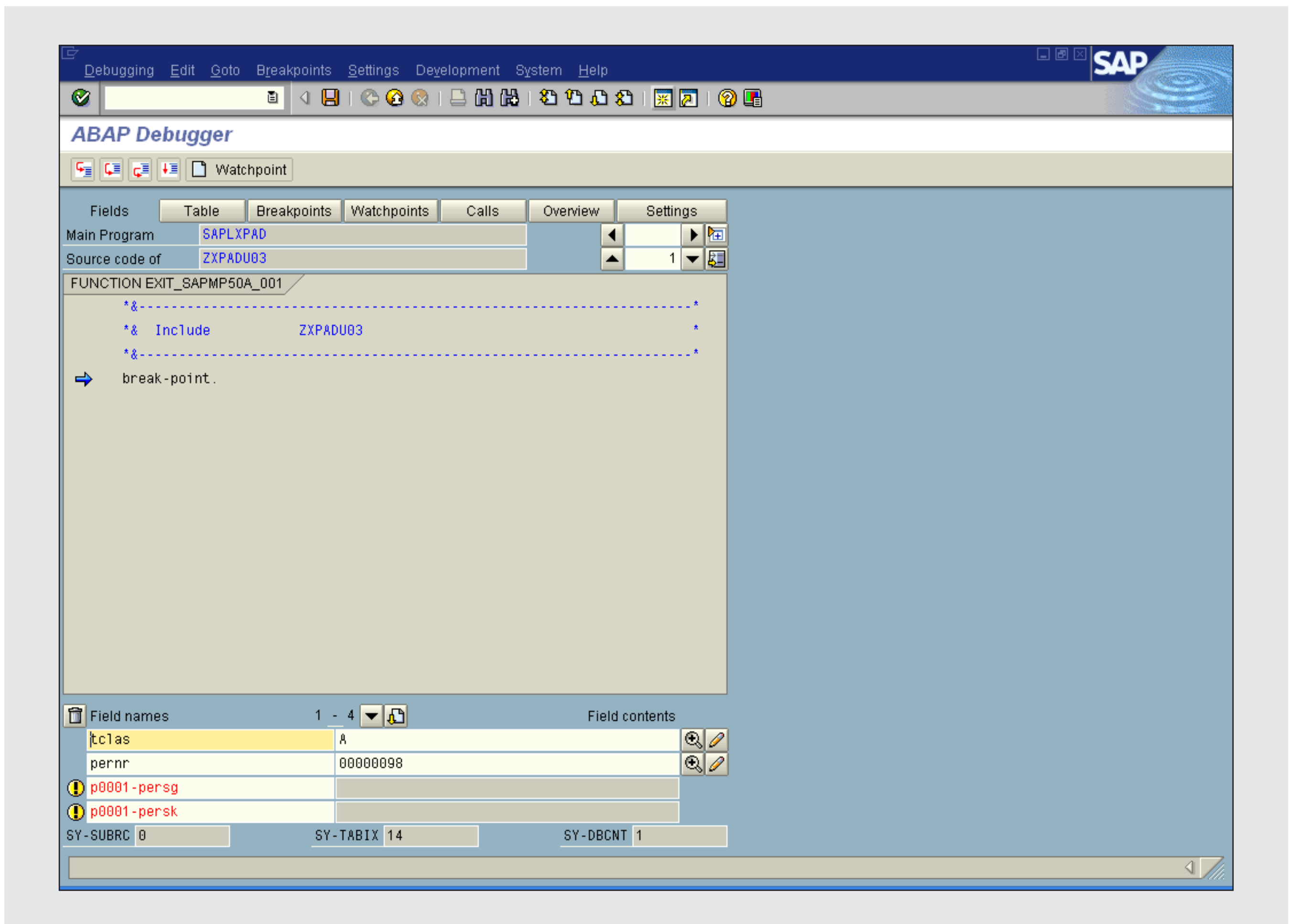

Figure 9 Debugger breakpoint within the user exit function module

is processed and the user exit function module is called. Using the Step Into button (or F5), execute the `CALL CUSTOMER-FUNCTION '001'` command.

Figure 9 shows that within the user exit function module, the employee group and employee subgroup fields are not accessible. Since they were not defined as parameters, they are not available, as is indicated by the exclamation mark icon and the red highlighting (p0001-persg, p0001-persk). **Figure 10** summarizes the parameters and their values.

 ©2008 SAP Professional Journal. All rights reserved.

Field	Value	Comment
pspar-tclas	A	Transaction class
rp50g-pernr	00000098	Personnel number
p0001-persg	-	Not available within user exit function module
p0001-persk	-	Not available within user exit function module

Figure 10 Field values within the user exit function module

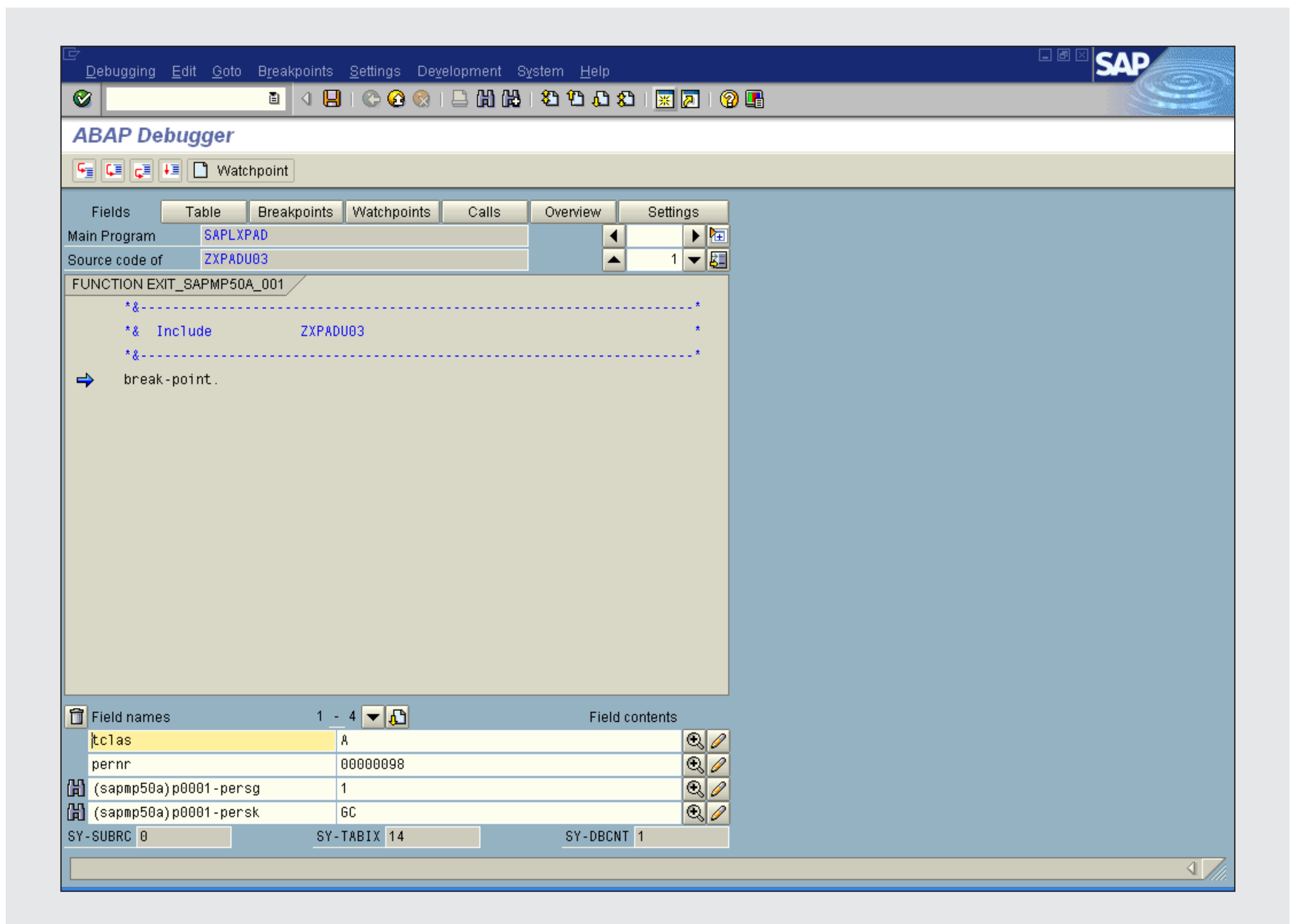

Figure 11 Displaying a global field within the user exit function module in the ABAP Debugger

The crucial point is this: It is possible to access these global fields provided they are defined correctly. All you need to do is enter the program name in parentheses before the field name in the field display (i.e., it should be formatted as *(main_program)global_structure*). Note that the program name is in parentheses. For example, the employee group p0001-persg is defined in conjunction with the main program sapmp50a, giving (sapmp50a)p0001-persg.

In **Figure 11**, the Field names window shows that it is still possible to read the value of these previously

Field	Value	Comment
pspar-tclas	A	Transaction class
rp50g-pernr	00000098	Personnel number
(sapmp50a)p0001-persg	1	Shows the value of the field in the main program
(sapmp50a)p0001-persk	GC	Shows the value of the field in the main program

Figure 12 Extended field values list within the user exit function module

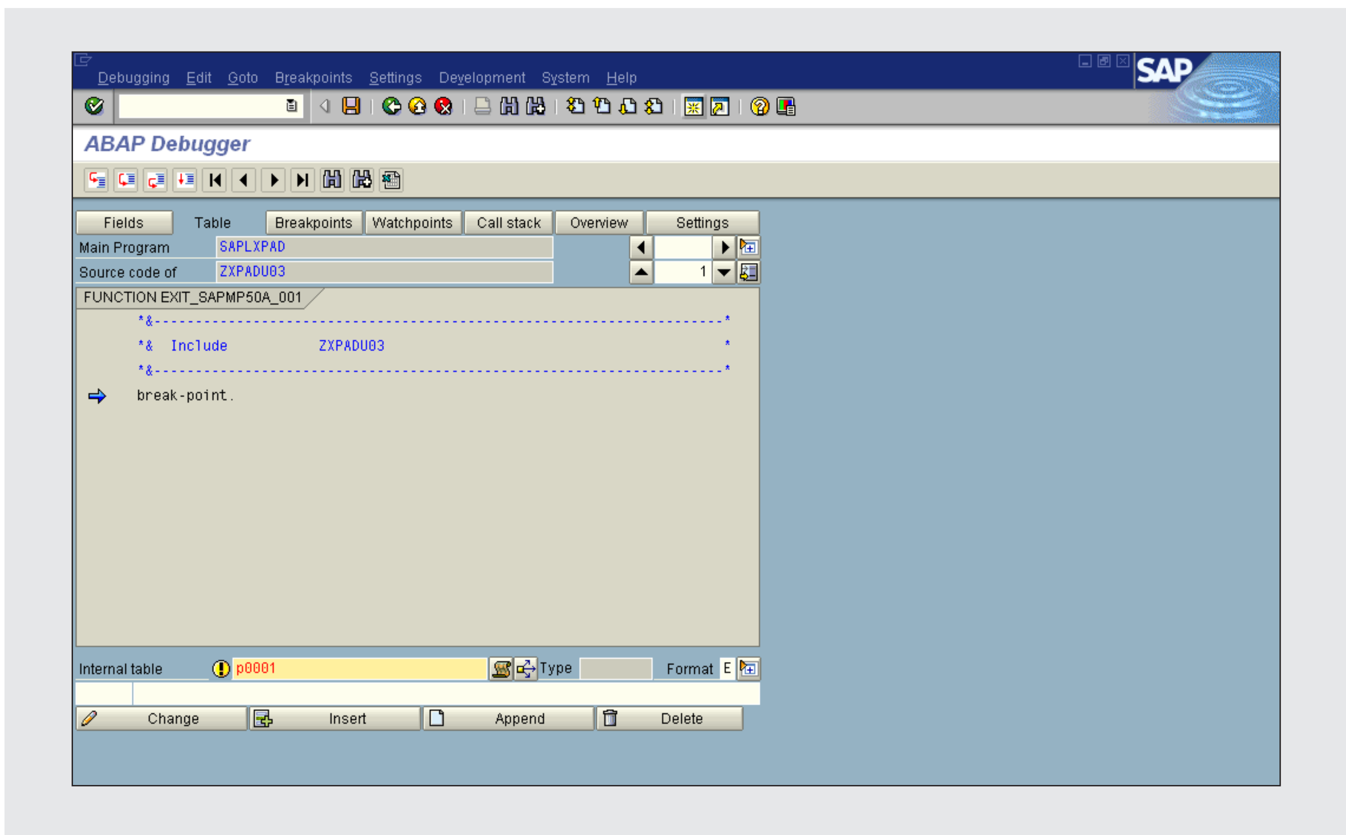

Figure 13 Internal table p0001 is not available in the user exit function module

unavailable fields inside the user exit function module, provided they have been identified this way. The field values list in **Figure 12** clarifies this.

The same is also true of any internal tables or structures that are available in the main program. After clicking the Table button on the ABAP Debugger (**Figure 11**) and entering the value "p0001," you find, as expected, that this table is not accessible from within the function model (see **Figure 13**). The exclamation mark next to the table name p0001, the table name highlighted in red, and the message bar showing "Specified table name not recognized" all indicate this.

 ©2008 SAP Professional Journal. All rights reserved.

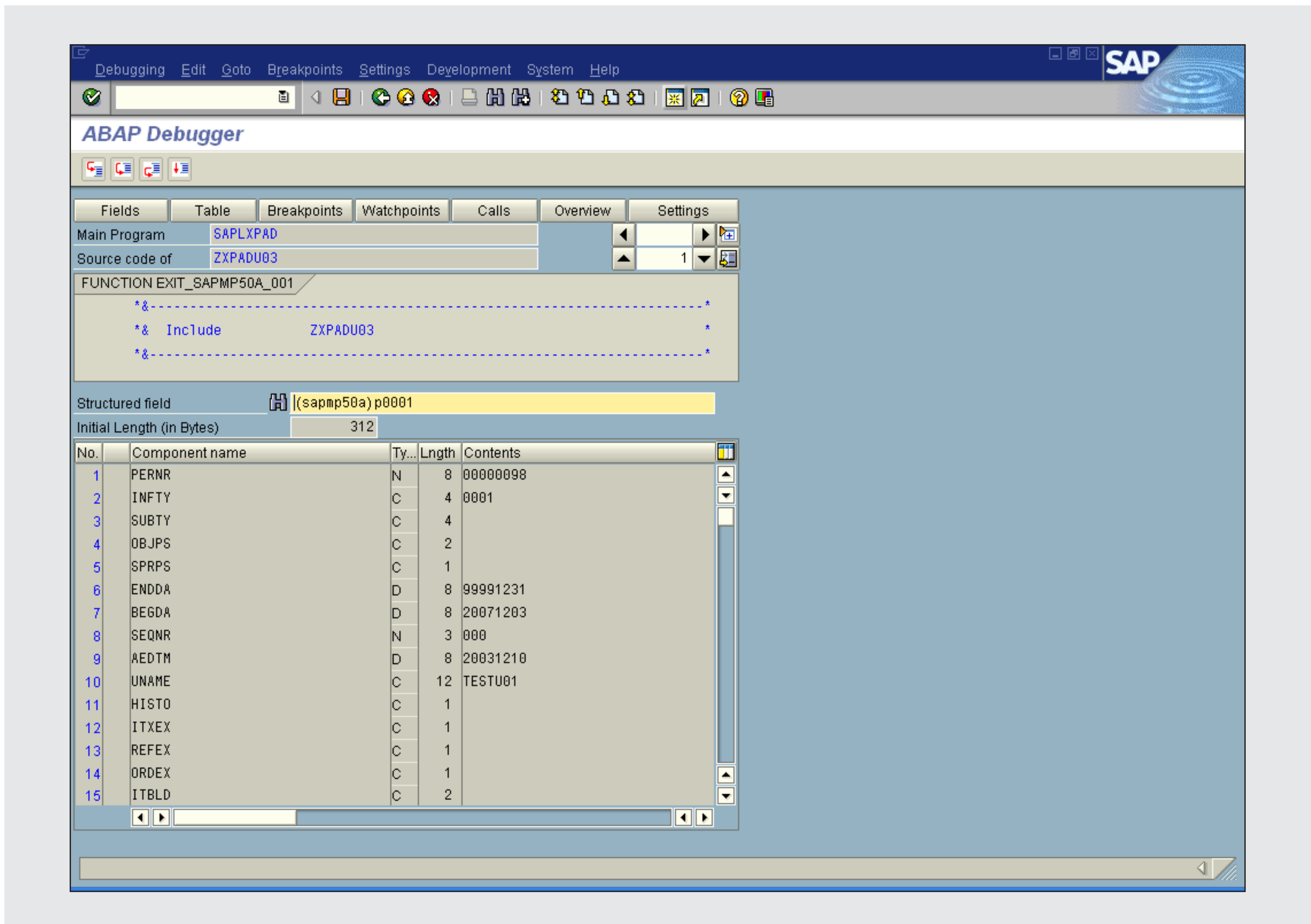

Figure 14 Displaying a global structure in the user exit function module

However, using the same approach as before, you can view the table if you define it in conjunction with the main program. Name the internal table field the same as the main program — in this case, (SAPMP50A)p0001. In **Figure 14**, the value "(SAPMP50A)p0001" appears in the structured field, and the window below shows that the individual components of the structure are now available. (Note that in this case, p0001 refers to a structured field with only one row, and not an internal table with multiple rows. However, you'll see how to access internal tables in the section "Sample code to read an internal table in the main program," which begins on page 69.)

The next step is to access fields in the main program by using ABAP commands instead of the debugger field and table windows. You simply use a type of variable known as a *field symbol*. (For an introduction to field symbols, see the sidebar on page 68.) You can dynamically assign a field symbol so it points to another variable, although you need to know the name of the variable. The process of dynamic assignment makes the field symbol actually refer to the contents of the variable. In addition, the dynamic assignment can evaluate the program that a variable has been defined by. In the HR example, by joining the main program name and variable together, the field symbol can be made to point to a variable in the main program. The short routine shown in **Figure 15** (on the next page) has been added to include ZXPADU03 to show how this is done. This has been set up to read the contents of the

p0001 structure in the main program, and the result has been passed to a local copy of the structure called local_i0001.

You can add a breakpoint to the routine and add the variable local_i0001 to the field names watch window, as shown in **Figure 16**. You can see that the correct values of the fields in p0001 are shown, indicating that it is possible to access this structured field in the main program from within the user exit function module, even though this has not been declared as a parameter.

All the fields held in structure p0001 are now available within the user exit function module. You can write the code to make decisions on the combination of employee group, subgroup, and payroll areas as required. You have now extended the functionality of the user exit function module and can use it to meet the customer requirement.

Sample code listings

This section presents simple component code listings that can be used as templates. Each example, then, clarifies how to access a field or an internal table in the main (or calling) program. However, there are no restrictions on combining both of these functions in one user exit or on the number of fields or tables that can be made available from the calling program within the same user exit.

> ***Note!***
>
> For each of the following code samples, replace all text in blue with your own specific data and program references.

Sample code to read a field or structure in the main program

You can use the code shown in **Figure 17** within a user exit when access is required to a variable or structure from the main calling program that isn't formally available as a defined parameter.

Explanatory notes

- Line 1: Defines a text string for the fieldname.

```
data:   structurename(40)  type c,
        local_i0001 like p0001.
field-symbols:  <g> type any.

structurename = '(SAPMP50A)p0001'.
assign (structurename) to <g>.
if sy-subrc = 0.

* contents of structure p0001 are passed to local structure table
  local_i0001 = <g>.

endif.
```

Figure 15 Sample code to read a structure in the main program

 www.SAPpro.com ©2008 SAP Professional Journal. All rights reserved.

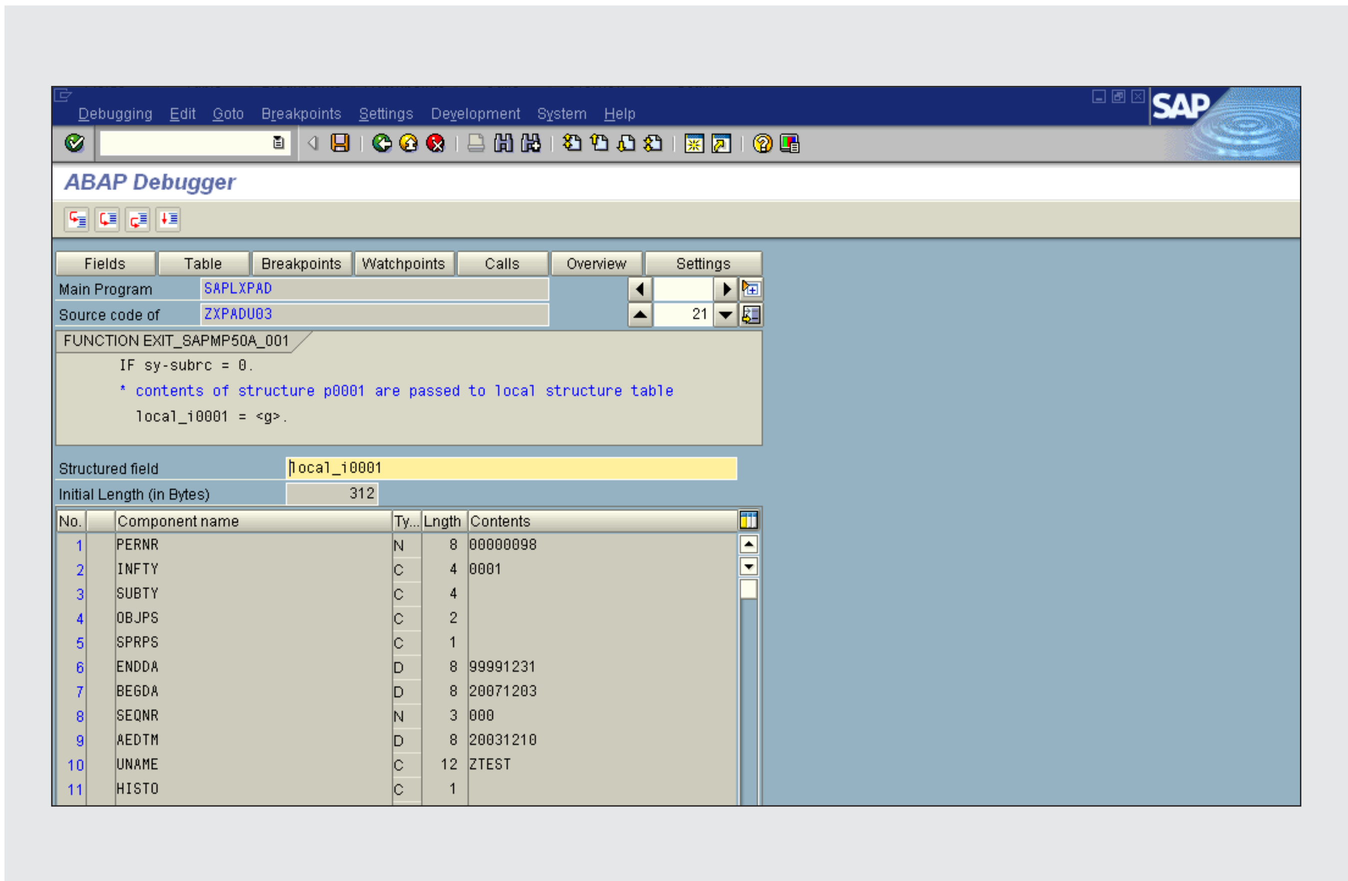

Figure 16 Displaying a global structure derived using field symbols in the user exit function module

```
1     data:   fieldname(40)  type c,
2     local_persg like p0001-persg,
3     field-symbols:  <f> type any.
4     fieldname =  '(SAPMP50A)p0001-persg'.
5     assign (fieldname) to <f>.
6     if sy-subrc = 0.
7     local_persg = <f>.
8     *.followed by further processing using local_persg
9     endif.
10     unassign <f>.
```

Figure 17 Sample code to read a field or structure in the main program

- Line 2: Defines a local copy of the data field p0001-persg containing the employee group that you want to use in the user exit but which is only available in the main program.

No portion of this publication may be reproduced without written consent.

Field symbols

The method of accessing missing parameters outlined in this article is possible because of the unique properties of field symbols. A detailed understanding of field symbols and all their applications is not necessary, since the code samples given below can be used as templates for your own development. Nevertheless, it's worth taking a moment to understand the background of field symbols to appreciate some of the considerations associated with using this method.

The best way of understanding field symbols is to compare them with "normal" variables in ABAP. A normal variable reserves a certain amount of space that a program can use as a *rewritable temporary store* for information. In contrast, field symbols are symbolic placeholders that represent other variables. They don't reserve any space in memory themselves but are used, rather, to access other variables. In coding terms, a field symbol is assigned to a variable, and from that point onward the field symbol can be used to access the contents of that variable. It is then possible to read or modify the contents of the variable using the field symbol.

This seems like a lot of work to achieve the same end result as setting up a normal variable in the first place. However, field symbols are useful to the programmer because they provide a flexibility that using variables on their own does not:

- The program can choose dynamically the data object that is assigned to the field symbol instead of the programmer explicitly specifying the name of a table or field to be read. Using field symbols means that the code can determine which field or table is read.
- Field symbols can be used to access any type of data object: a variable, string, or internal table. One field symbol can point to different data types.
- They can have different technical attributes from the assigned variable, thereby allowing the programmer to avoid incompatibility of data types or other syntax restrictions.

This flexibility allows for some elegant solutions to difficult problems. Field symbols are commonly used to dynamically access entire tables or to loop through all the fields in a structure to read or modify each one in turn. Using field symbols for intensive processing of internal tables can significantly improve performance over other programming methods. (Examples can be found in many ABAP developer forums.)

The disadvantage of using field symbols is that code can be more difficult to understand and debug since it isn't immediately clear to what a field symbol refers. Syntax checking within ABAP may not pick up on errors until runtime. In general, and where possible, error-checking should also be made more robust to cope with the increased variety of errors that may be encountered.

For the purposes of this article, however, it is the dynamic assignment property of field symbols that is of interest. From within the user exit, you can define the field symbol to point to a variable in the main program. At its simplest, all that is required is to pass the name of the calling program and the name of the variable into a text string. When this is dynamically assigned to a field symbol, the text in the string is evaluated so that the field symbol will point to the actual calling program and variable.

 ©2008 SAP Professional Journal. All rights reserved.

> ***Note!***
>
> Field symbol names must always begin and end with angle brackets (< >).

- Line 3: Defines a field symbol named <*f*>.
- Line 4: A string with the name of the calling program in parentheses, followed by the full field name, is passed into the fieldname string.
- Line 5: The fieldname string is dynamically assigned to the field symbol. Immediately after this line executes successfully, the field symbol will point to the field in the calling program.
- Line 6: Checks the return code immediately after the assign command. The return code sy-subrc is checked to confirm that the field string was assigned without any error which might occur if the program or field name were incorrectly defined. If an error occurs and the dynamic assignment cannot take place, then the variable will not be read successfully and the routine will not work. The assignment statement must always be checked for success to trap this possibility.
- Line 7: The contents of the field symbol are passed to a local field. At this point, the local field has the same contents as the field in the main program and can be used in the same way as any of the other parameters in the function. Since this is a local copy, it will contain the same value as the field in the calling program; but the value can be changed or manipulated without affecting the original field in the main program.
- Line 8: Further processing using the local field should be inserted here. For example, a decision could be made on the employee subgroup in the local field using either an IF or a CASE statement.
- Line 9: All code that should only be carried out if the field symbol is assigned correctly should be put before this ENDIF statement.
- Line 10: The field symbol is cleared using the unassign command.

Sample code to read an internal table in the main program

With a slight modification to the code in **Figure 17**, you can enable your program to read all the data stored in an internal table. You can use the code (as shown in **Figure 18**) in a user exit when access is required to an internal table from the main program that hasn't been declared in the function module interface.

```
1    data:   tablename(40)  type c,
2    local_i0001 type table of p0001.
3    field-symbols:  <f> type any table.
4    tablename = '(SAPMP50A)p0001[]'.
5    assign (tablename) to <f>.
6    if sy-subrc = 0.
7    local_i0001 = <f>.
8    * ...further processing of internal table
9    endif.
10   unassign <f>.
```

Figure 18 Sample code to read an internal table in the main program

No portion of this publication may be reproduced without written consent.

Explanatory notes

- Line 1: Defines the text string for the table name.
- Line 2: Defines a local version of the table that you want to use in the user exit that is only available in the main program.
- Line 3: Defines a field symbol named <*f*> with a type specification that the data structure to be assigned is a table. This ensures that the field symbol will inherit all of the attributes (line type, table type, key) from the data object.
- Line 4: The tablename string is given a value of the name of the calling program in parentheses followed by the table name and square brackets. *The square brackets after the table name must be present to pass all the table entries to the user exit.*
- Line 5: The table name is dynamically assigned to the field symbol, which now points to the internal table in the calling program.
- Line 6: Checks the return code immediately after the assign command. The return code sy-subrc is checked to confirm that the field string was assigned without error. An error might happen if the program or field name had been incorrectly defined. If there is an error, then the routine will not work.
- Line 7: The contents of the field symbol are passed to a local table. At this point the local table local_i0001 has the same contents as the table p0001 in the main program and can be used in the same way as any of the other parameters in the function. The table local_i0001 can only be accessed within the user exit function module; changes made to local_i0001 do not affect table p0001 in the calling program, and table local_ i0001 is cleared as soon as the user exit function module has been run.
- Line 8: Further processing using the local field should be inserted here.
- Line 9: All code that should only be carried out if the field symbol is assigned correctly should be put before this ENDIF statement.
- Line 10: The field symbol is cleared using the unassign command as before.

> ***Note!***
>
> If you accidentally use the method to derive a field instead of the method shown here to derive a table, this will be interpreted as referring to the header line of the table, and not the table body.

In the example given here, an alternative approach to using the field-symbol method would have been to derive the employee group from first principles. For example, you could have used an infotype table read statement to read all the infotype 0001 values into an internal table. Assuming that the relevant start date could also be derived within the user exit, it would then be possible to select the appropriate record (e.g., current infotype 1 record) and then determine the employee group. However, this approach has several disadvantages:

- **Redundant code:** All the data that is read directly from database tables must be error-checked and validated. Further data conversion or calculation that has already taken place within the main program may also need to be repeated, leading to duplication of code within the user exit.
- **Incorrect processing:** Any data conversion or calculation that is repeated in a user exit might produce a different result from the original processing in the main routine, leading to errors.
- **Reduced error-trapping:** Within the user exit, it might not be possible to correctly trap or log errors and successfully pass these back to the main routine.
- **Support packs and SAP (OSS) Notes:** There is an increased overhead for support whenever SAP modifies a program in a support pack or SAP Note. If the main program or routine is modified

  ©2008 SAP Professional Journal. All rights reserved.

significantly, the user exit that repeats this routine also needs to be modified manually.

Therefore, even in cases where it is possible to read the database to derive the missing data, the field symbol approach may provide a simpler and superior solution.

In many cases it is not possible to read directly from a database table. This may be because the required parameter was never written to a database table. Alternatively, it may be that the parameter has been modified in the main program and is in an intermediate stage of processing at the point that the user exit function module is called; since the calling program has not updated the database table, it will not be possible to implement a read in this case either.

Sample code to modify a field in the main program

The code shown in **Figure 19** demonstrates that it is also possible to modify the contents of a field in the main program from within a user exit function module, even if this is not passed as a parameter.

> ***Note!***
>
> Modifying the values of fields in the main program that have not been passed to the user exit function module as parameters can cause program errors and other unpredictable responses. It should only be used when you are absolutely sure that you understand the "downstream" effects in all circumstances. *Make sure you have read and understand this entire section before attempting this method.*

Explanatory notes

- Line 1: Defines text string for the field name.
- Line 2: Defines a local structure based on a structure in the main program.
- Line 3: Defines a field symbol named <*f*> without any type specification.
- Line 4: A string with the name of the calling program in parentheses, followed by the table name, is passed into the field name.

```
1    data:   fieldname(40)  type c,
2    local_structure like mainprogram_globalstructure.
3    field-symbols:  <f> type any.
4    fieldname = '(main_program)global_structure'.
5    assign (fieldname) to <f>.
6    if sy-subrc = 0.
7    local_structure = <f>.
8    * ... further processing of local_structure
9    * now the modified values are passed back to the calling program
10     <f> = local_structure.
11   endif.
12   unassign <f>.
```

Figure 19 Sample code to modify a field in the main program

No portion of this publication may be reproduced without written consent.

- Line 5: The field name is dynamically assigned to the field symbol, which now points to the global structure in the calling program.
- Line 6: Checks the return code sy-subrc immediately after the assign command. The return code is checked to confirm that the field string was assigned without error; an error might happen if the program or field name had been incorrectly defined. If there is an error, then the routine will not work.
- Line 7: The contents of the field symbol are passed to the local structure.
- Line 8: Further processing using the local structure is inserted at this point. Changes can be made to the values within the structure.
- Line 10: The values in the modified structure are now passed back to the field symbol <*f*>. Because this still points to the global structure in the main program, the value stored in the main program will be updated.
- Line 11: All code that should only be carried out if the field symbol is assigned correctly should be put before this ENDIF statement.
- Line 12: The field symbol is cleared using the unassign command.

These code listings show you how to use field symbols to extend the range of data objects available in a user exit function module. Any field or structure in the main program can be made accessible by using a dozen or fewer lines of code. With a slight modification to the method, internal tables can also be read. A final code listing shows that it is also possible to update variables in the main program; however, I have pointed out that this could cause serious problems and errors. You must exercise extreme caution, and this last method should only be considered as the means of last resort.

General considerations

This technique is undoubtedly useful for extending functionality. However, it can also be risky in certain circumstances. So there are several considerations of which you should be aware.

First of all, this method should always be considered from the viewpoint of good programming technique. Some programmers might object to the use of this technique on the grounds that it violates the separation of global and local variables, that this method contravenes the principles of modularization within programming. Beyond this, in the context of SAP, it might be argued that the parameters passed in a user exit have been carefully considered and are "protected" in the sense that they will not be changed when a support pack is applied. This guarantee may not extend to the extra variables or tables obtained using this method.

These are all valid points and should not be taken lightly or dismissed. They are clear indications that this technique should be reserved only for those situations where the alternatives are even less acceptable and involve far more work to achieve the same result.

If this is the case, there are a number of actions that should be taken to minimize any risks associated with using this method.

Testing

Clearly, the testing procedure should be more rigorous than normal. As mentioned above, the ABAP syntax checker cannot fully verify that a field symbol will be assigned correctly until the program runs. This means there is a greater risk of unforeseen errors than with normal code. Errors of this nature can lead to an abnormal termination (abend) if not trapped.

It is vital that you check all possible situations in which the user exit is called. Some user exits are called from more than one program. You can check this by using the Where-Used List button on the user exit screen. In this case, you will need to check that the modifications produce consistent results as required. Note that when a user exit is called by different programs, it is possible to have the user exit respond in different ways depending on which program has called it. This can be achieved by using a case statement based on the technical name of the calling program stored in the system field sy-repid. You can also use a similar method to differentiate between background and foreground processing by using sy-batch.

 ©2008 SAP Professional Journal. All rights reserved.

Caution!

This code listing shows how it is possible to change the value of a field in the main program from within a user exit, even if that field has not been passed as a parameter. It cannot be stressed strongly enough, however, that this approach is highly risky and is not recommended. The main program will only be expecting changes to have been made to the export parameters of the user exit function module. After the user exit has been called, the main program will almost always check these values for validity to trap any possible errors. In contrast, any changes made to variables in the calling program (which are not formally declared as parameters) using the field symbol method will not be checked. This could lead to data inconsistencies or other program errors. Examples of the problems this might cause include:

- An immediate program termination (abend)
- Data could become corrupted or unreliable in a manner that is neither obvious nor immediate
- Database relational integrity could be compromised

Clearly, these are all serious problems. In addition, the changed parameter may cause problems much later in the program execution if the call to the user exit function module is nested deep in the call stack. Even a seemingly innocuous change, such as sorting a table, can lead to disastrous results. In other words, it is not enough to simply check the status after the function module has exited. The entire lifecycle of the variable in question must be analyzed. This is not an easy task.

Nevertheless, the option to change any value within the main program exists. It is included in this article so that the full range of possibilities available is known. In the first instance, even if this method is never used, you should be aware that it exists and has consequences just so that you don't code it by accident. Beyond this, there may be some cases where its use can be justified. If it can be proved that changing a data object in this way will have no adverse consequences, then in certain carefully limited circumstances it might provide a solution where there are no other options or those that exist are even less palatable. If the only alternatives are either to create a completely customized version of a report or transaction or to add a modification to SAP standard code, then this may be the best option — provided you are aware of the potential risks. Consultants must carefully weigh their options and mitigate the risks to the best of their ability.

It is also important that no unchecked assumptions are made about the field structure or table that is being obtained from the main program. Since you have not derived this yourself, you need to be sure that it contains the information you need. For example, it might be necessary to obtain the employees cost center p0001-KOSTL. A quick investigation using a test employee may confirm that an infotype 0001 structure is available in the calling program. However, an employee may change position and therefore have multiple infotype 0001 records: Have you checked which of these is being used in the calling program? It may be the current record, or it may be the first record, or the first record valid at whatever selection date the main program uses. All these combinations must be checked to ensure that the relevant cost center is being correctly derived. This is why I have stressed that using the debugger and robust testing is essential.

Documentation

The use of this method should be well annotated in the programming comments and supporting

documentation. Aside from being good practice, this is important because support consultants may not be familiar with this technique. It is easy to foresee a situation, for example, where a consultant might be looking for a problem related to a particular field but would not think to look in the user exit since the field is not formally passed as a parameter.

Taking support packs into consideration

This approach requires more effort whenever a support pack is applied. SAP will ensure that the formal parameters supplied to a user exit are not changed, but this will not be the case for any fields that are derived using this method. Therefore, the user exit should be rechecked whenever the calling program is modified. Care should also be taken if the calling program is itself called by another program that may have been modified, because this may also signify that a significant change has been made.

Reviewing during upgrades

Normally, a user exit is "protected" in an upgrade and should continue to work as it did previously. However, any user exits making use of this method would need to be completely reviewed for the same reasons mentioned above. In my experience all user exit function modules are fully reviewed in an upgrade, regardless of whether they used this approach.

A final word of caution

Finally, and most importantly, you should not make any changes to the variables in the main program unless it is unavoidable. The examples have shown how local copies of variables in the main program should be used for making decisions or calculations. While possible, changes to the actual variables in the main program should not be made. As previously mentioned, even so seemingly innocuous a change as sorting a table can lead to problematic results. It would be well worth your time to re-read the section on modifying values so that you remain acutely aware of this and don't do it by accident.

Conclusion

The simple method and code extracts shown here can be used to access data fields in the main program other than those formally provided to a user exit function module. I strongly recommend that you use data read from the main program to make decisions or calculations within the user exit function module, and that updates should only be made to the formal output parameters available to the user exit. To minimize any risks, you should carry out a thorough analysis of the context in which the user exit is called and understand the program logic thoroughly.

Exactly the same method can also be applied to extend the functionality of SAP standard HR features. It is well-known that where the requirements are too complex for the normal tree-like decision structures, ABAP routines can be called from within the feature. However, the normal limitation remains that the only fields available are those that have been declared in the feature structure. Using the field symbols method, it is possible to extend the parameters available. For example, feature LGMST is used to determine a set of wage types that are defaulted on the infotype 8 basic pay screen. I have used this method to extend standard functionality of this feature and make complex decisions based on pay scale type and pay scale area that would not otherwise have been possible. I have also used this method within custom payroll functions and operations. This has enabled me to create simple, short, and powerful enhancements to the standard toolset provided by SAP for the payroll schema.

Provided the safeguards are followed, this method can be a very useful tool for extending the range of solutions available across a host of day-to-day consulting problems.

 ©2008 SAP Professional Journal. All rights reserved.