



EGL Programmer's Guide

Contents

Developing segmented programs in EGL 1

Running in segmented mode	1
Running in nonsegmented mode	2
Comparison of segmented and nonsegmented programs for CICS	2
Choosing between segmented and nonsegmented programs	5
Program design considerations	6
Implementing a hierarchical structure for segmenting programs using a transfer to program statement	7

Switching transaction codes for program segments . . .	9
Using a show statement with inputForm	11
Accessing multiple DB2 plans in z/OS CICS	12
Accessing multiple DB2 plans in IMS.	14
Error processing for segmented programs	14

Index	15
------------------------	-----------

Developing segmented programs in EGL

When you create an EGL Text UI program, you can specify whether the program is to run in segmented mode or nonsegmented mode by setting the **segmented** property. Based on the target runtime environment, the use of the **segmented** property affects both the generation of the program and how the program behaves at a **converse** statement.

Debug, Java generation, and COBOL generation for iSeries

EGL simulates the logical effects of segmented mode by committing recoverable resources, reloading library parts, refreshing certain system variables, and refreshing the contents of single-user DataTables at a segmented converse.

CICS Nonsegmented processing is equivalent to CICS conversational processing; segmented processing is equivalent to CICS pseudoconversational processing.

IMS/VS

Text UI programs must always run in segmented mode.

Related information

“Running in segmented mode”

“Running in nonsegmented mode” on page 2

“Error processing for segmented programs” on page 14

Running in segmented mode

When a converse occurs while running in segmented mode, a program saves current program status in a work file or database. An example of current program status is the current values for variables. The program releases all storage, file, and database resources whenever it requests input from the terminal using a **converse** or a **show** statement. When running in nonsegmented mode, these resources are not released.

Before defining programs that run in segmented mode, you must understand the effect of segmenting in the runtime environment. Because segmenting might alter the results of the program, you must consider whether or not to segment programs during the initial design phase.

Segmented mode enables a larger number of terminals to run EGL programs within the same system storage address space for CICS systems at the same time. Although segmenting programs enables concurrent use by a larger number of terminals, the response time for each terminal is increased by the time required for each transfer of data (roll out or roll in), and the time required by the host subsystem to create a new system task.

Segmented programs do not use address space during user think time. This is because the program address space is saved on external storage when the current system task ends, and the program needs input from the user to continue.

When the user presses Enter, Clear, a PA key, or a function key, a new system task is started. EGL restores the address space for the task with the data retrieved from external storage.

Related information

“Running in nonsegmented mode”

“Comparison of segmented and nonsegmented programs for CICS”

Running in nonsegmented mode

Nonsegmented programs consume address space from start to finish including user think time. User think time starts with each **converse** statement and varies by program and form. System resources are used by the program while waiting for the user to enter the next transaction, as when the user presses **Enter**, **Clear**, a PA key, or function key.

Related information

“Running in segmented mode” on page 1

“Comparison of segmented and nonsegmented programs for CICS”

Comparison of segmented and nonsegmented programs for CICS

For CICS environments, you can specify whether a program runs in segmented (CICS pseudoconversational) or nonsegmented (CICS conversational) mode by setting the **segmented** property when you define the program. If you are running in VAGen compatibility mode, you can also dynamically change the runtime mode for the program by using the **converseVar.segmentedMode** system variable. **converseVar.segmentedMode** is set to the default value (1 for segmented mode and 0 for nonsegmented mode) after every **converse**.

The following diagram illustrates the flow of a program running in nonsegmented mode. The sample update program, CSUP, converses a form, displays the customer data that can be updated, accepts data from a user to update the customer record, and replaces the record with the changed data. When you use nonsegmented mode, you should set the **converseVar.commitOnConverse** system variable to 1. This causes a commit point to occur at the **converse**, so that changes to files and databases are committed and locks are released. The diagram also illustrates saving a copy of the record for comparison purposes after the **converse** to ensure that no other changes have been made to the record during user think time.

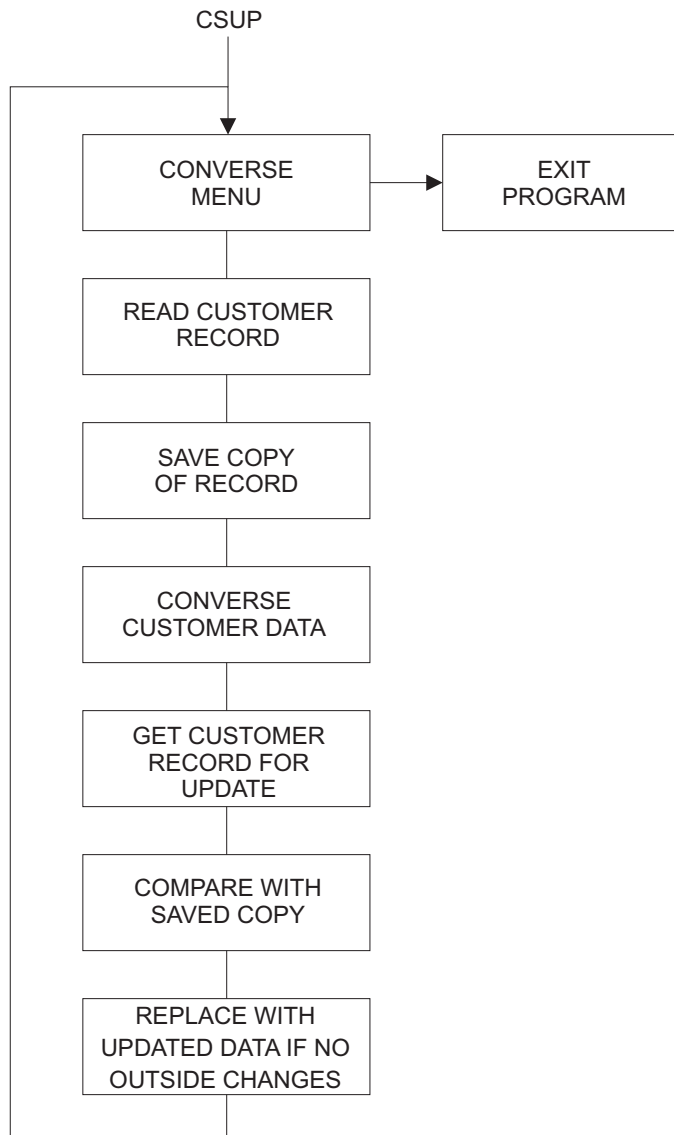


Figure 1. Update file program running in nonsegmented mode

The following diagram shows the flow of the same update program running in segmented mode.

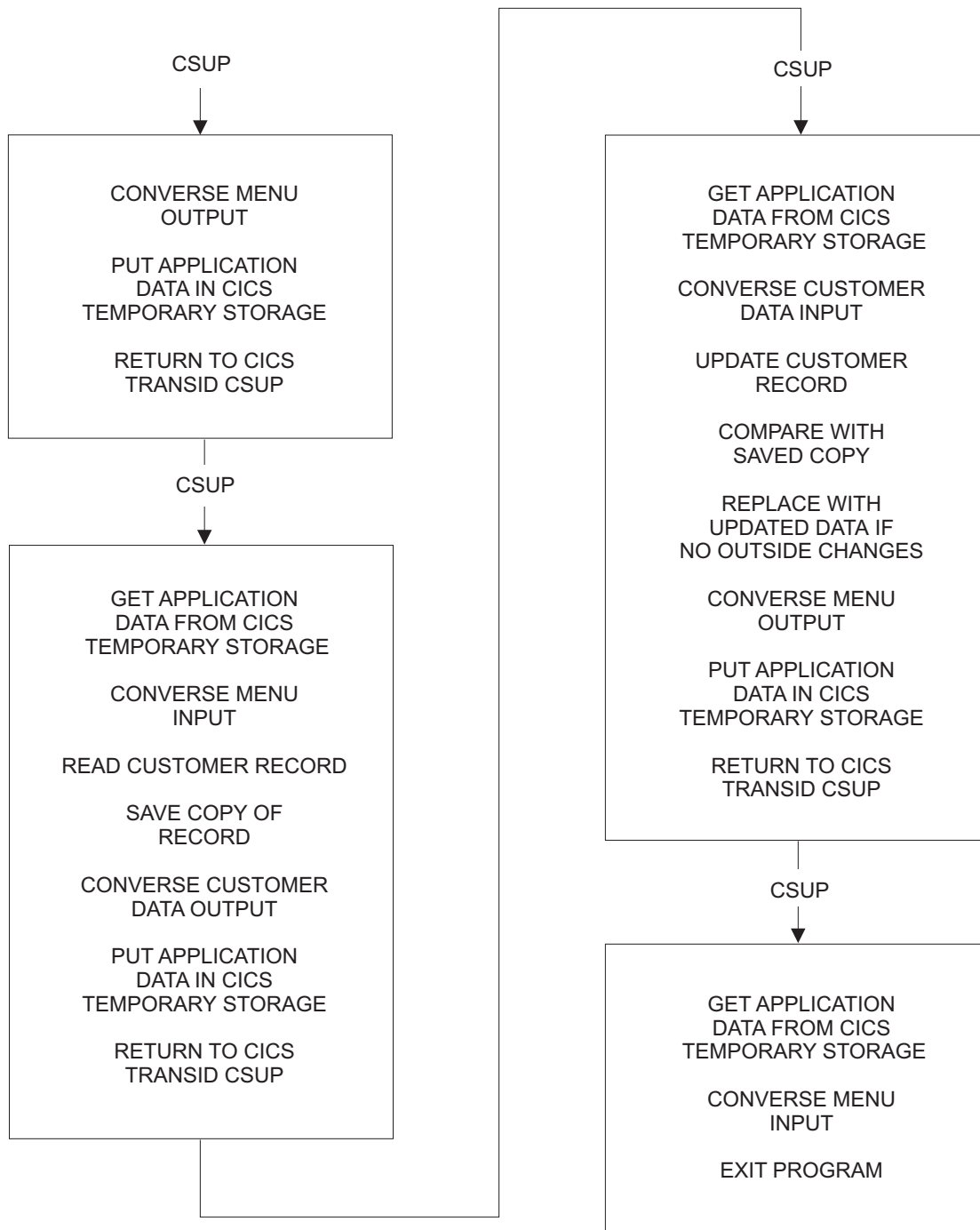


Figure 2. Update file program running in segmented mode

When a program runs in segmented mode, temporary storage must be provided to contain the roll out/in data during segmentation. Each program requires approximately 6000 bytes plus the total size of all objects accessed by the program (records, variables, and forms). In addition, because the program must be started after each user input, you might want to make the program, print services program, and FormGroup format module resident for segmented programs.

Related information

“Running in segmented mode” on page 1

“Running in nonsegmented mode” on page 2

“Choosing between segmented and nonsegmented programs”

Choosing between segmented and nonsegmented programs

When you are deciding whether to design your program as segmented or nonsegmented, you should be aware of two issues. The first issue is the effect of the transaction on contention resources, such as storage and processor use. The second issue is the effect on exclusive-use resources, such as records and recoverable data sets, recoverable transient data queues, and enqueue items.

Nonsegmented programs have a high impact on storage because they run longer than the sum of the transactions that are in an equivalent segmented program. However, processor overhead is lower because only one program is started instead of one for every transaction.

A nonsegmented program retains exclusive use of resources for a longer period of time, unlike the equivalent segmented program. For this reason, segmented programs are quicker to respond, but for recovery and integrity considerations, you might prefer a nonsegmented program.

If you have forms in called programs or need to lock the database during a **converse**, you should design your program to run in nonsegmented mode.

The following list contains considerations for segmented and nonsegmented programs:

- Segmented mode uses more processor time because CICS spends more time initiating and ending transactions.
- Nonsegmented mode uses more virtual storage because transactions are still active during user think time. However, with Dynamic Transaction Routing (CICS/ESA 3.1) CICS can automatically start another region and send transactions to the next region when the first region is constrained.
- Nonsegmented mode can also use other resources such as locks in the database during user think time. (This can be solved by setting the system variable **converseVar.commitOnConverse** to 1.)
- CICS accounting and security is less granular with nonsegmented transactions because you have a few large transactions, rather than a lot of small ones.
- CICS shutdown can be more difficult with a lot of nonsegmented transactions. You might have to end transactions before you can shut down because someone is out for a break in the middle of a nonsegmented transaction. (This can be solved by having transactions time out if the user has not pressed **Enter** after a specified time).
- Programming nonsegmented programs can be easier because you can do the following:
 - Use text forms in called programs.
 - Hold locks and cursor position in the database over a **converse**.
- Only segmented programs can be generated for the IMS environment.

When running a program on a system where storage contention is not a problem, a good compromise between running in segmented or nonsegmented mode is to run in nonsegmented mode with the **converseVar.commitOnConverse** variable set to 1.

This approach forces a commit at every **converse** from the main program, and it has the good performance characteristics of nonsegmented mode, but it does not hold file or database resources during user think time.

Related information

“Running in segmented mode” on page 1

“Running in nonsegmented mode” on page 2

“Comparison of segmented and nonsegmented programs for CICS” on page 2

“Program design considerations”

You should consider a number of factors when designing segmented programs.

Program design considerations

You should consider a number of factors when designing segmented programs.

- Any called program loses the return point if segmentation occurs; therefore, the following restrictions apply:
 - EGL programs defined as called transactions can run **converse** statements that reference forms but cannot be generated to run in segmented mode. Called Text UI programs are not supported for the IMS environment.
 - If a program is called from a main program that is running in segmented mode, the transaction runs in CICS conversational mode (nonsegmented) until the program returns from the called program.
 - If an EGL program calls a non-EGL program, the non-EGL program cannot use segmented mode. Any interaction with the program user must be in CICS conversational (nonsegmented) mode.
- Segmentation ends the current system task. CICS and IMS commit all recoverable resources when the task ends.
- A record cannot be held for update (locked) across a segmented **converse**.

Note: Holding a record for update across a **converse** is not a good practice on any system, because it locks resources during user think time, preventing additional users from accessing the system.

For a better approach to holding a record for update across a **converse**, refer to the code in the following example.

```
customerRecord Customer;  
savedRecord Customer;  
updateComplete char(1) = "N";  
  
// check that data has not changed during user think time  
customerRecord.CustomerID = 1;  
get customerRecord;  
while (updateComplete == "N")  
    move customerRecord to savedRecord byName;  
    move customerRecord to custDetailForm byName;  
    converse custDetailForm;  
    // validate input data on custDetail form  
    // assuming validation passed, continue  
    get customerRecord;  
    // check all fields in customerRecord to determine  
    // whether anything changed during user think time  
    if (customerRecord.field1 == savedRecord.field1  
        && customerRecord.field2 == savedRecord.field2  
        ...  
        && customerRecord.fieldn == savedRecord.fieldn )  
        // if no changes, move changed data from form to customerRecord  
        replace customerRecord;  
        updateComplete = "Y";  
    else
```

```

        // message to user that data was modified by someone else
    end
end

record Customer type ...
    field1 ...
    field2 ...
    ...
    fieldn ...
end

```

- Locks created by the **forUpdate** keyword and current positions in files or databases are lost during a **converse** statement when running in segmented mode.
- The program structure and I/O objects determine the amount of response time delay caused by the roll out/roll in process:
 - The longest delay occurs in a segmented program that has a large amount of variable field data on forms or large records and short user think time.
 - The shortest delay occurs in a menu type program that has a small amount of variable field data on forms, only a small record, and long periods of user think time.
- In CICS, if the UCTRAN operand has been set to YES for the terminal control table PROFILE or TYPETERM entries for the terminal, CICS folds user data from forms to upper case when running in segmented mode. The folding of user data by CICS causes the EGL **upperCase** property have no effect.
- On CICS systems, when the user presses the **Enter** key or a function key, the system returns input data through CICS to the EGL program. CICS examines the beginning of the data, searching for Basic Mapping Support (BMS) commands. When designing segmented programs, ensure that the first physical variable field on your EGL form does not contain a valid BMS paging command. For more information on design considerations for segmented programs in CICS, refer to the CICS documentation.

Related information

“Implementing a hierarchical structure for segmenting programs using a transfer to program statement”

This section describes a set of EGL programs that should perform well running in segmented mode in the CICS environment.

Implementing a hierarchical structure for segmenting programs using a transfer to program statement

This section describes a set of EGL programs that should perform well running in segmented mode in the CICS environment.

An additional benefit is that these programs are easier to maintain, test, and enhance than a single EGL program that contains all the same functions.

The diagram below shows the relationship between four EGL programs that perform three tasks. The only purpose of the menu program is to access the other three programs. When the user selects the desired task, the menu program transfers control to the corresponding program using a **transfer to program** statement. The menu program passes a small working storage record to further define the request. The transferred-to program prompts the user for required data, performs the task as often as needed and uses the **transfer to program** statement to return to the menu program.

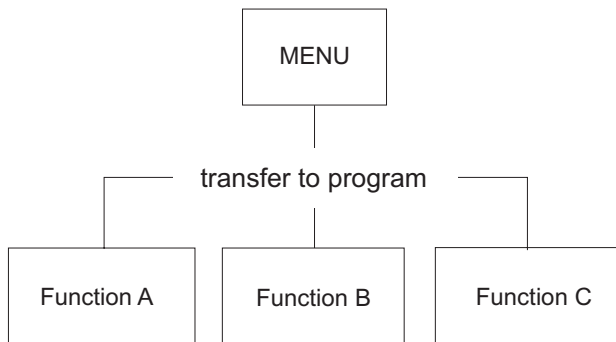


Figure 3. Hierarchical structure using a transfer to program statement

Using a **transfer to program** statement solves the following segmented mode restrictions:

- Called programs cannot run in segmented mode.
- Called programs do not release the caller's resources because program control returns to the calling program.
- The amount of data rolled in/out during a segmented converse is much smaller. Only the program currently in control has its data areas saved.

In the IMS/VS environment, a **transfer to program** does not release the storage for the original program (the MENU program in the example). If you are developing programs that run in both IMS and CICS environments, you can use either a **transfer to program** or a **transfer to transaction** statement. If you are developing programs that run in IMS only, use a **transfer to transaction** statement for the following reasons:

- To free resources for the transferred-from programs
- To cause a commit point and release **forUpdate** locks
- To permit each program to have its own DB2 plan and a different PSB
- To permit each program to have different performance tuning information in the IMS system definition

Use a **transfer to program** statement in the IMS environment if you do not want a commit point to occur or if you need both programs to use the same DB2 plan and PSB.

Dynamically changing between segmented and non-segmented mode

Note: This technique is only supported in VAGen Compatibility mode. It should not be used for new programs.

When you specify the **segmented** property for a program, you set the default mode for how a **converse** statement is handled at runtime. The **segmented** property initializes the value of the **converseVar.segmentedMode** system variable in the following way:

- 0 indicates that the program is nonsegmented
- 1 indicates that the program is segmented

You can change the **converseVar.segmentedMode** system variable to dynamically control segmentation at runtime. By setting **converseVar.segmentedMode** to 0 or 1, you can override the default value for the next **converse** statement. Before each

converse statement, EGL checks the value of **converseVar.segmentedMode** and processes the **converse** statement in the following way:

- If **converseVar.segmentedMode** is set to 1, EGL treats the **converse** statement as a segmented converse.
- If **converseVar.segmentedMode** is set to 0, EGL treats the **converse** statement as a nonsegmented converse.

When the **converse** statement completes successfully, EGL resets **converseVar.segmentedMode** to the default value based on the **segmented** property for the program.

The **converseVar.segmentedMode** system variable enables you to switch in and out of segmented mode for reasons of performance, function, and target system differences. To control segmentation, use either of the following statements prior to a **converse** statement:

```
converseVar.segmentedMode = 1; // force the next converse to be segmented
converseVar.segmentedMode = 0; // force the next converse to be nonsegmented
```

Remember that **converseVar.segmentedMode** is reset to its generated default after every **converse**. Therefore, you should only set **converseVar.segmentedMode** if you want to alter the behavior of a specific **converse** from the default processing.

Note: EGL ignores the use of **converseVar.segmentedMode** system variable in the IMS/VS environment. All programs containing **converse** statements must run in segmented mode in the IMS/VS environment.

Related information

“Program design considerations” on page 6

You should consider a number of factors when designing segmented programs.

“Switching transaction codes for program segments”

Switching transaction codes for program segments

On CICS and IMS systems, a segmented **converse** statement ends the current transaction. Terminal input from the user starts a new transaction. The new transaction is identified by the **sysVar.transactionID** system variable when the program processes the **converse** statement. The default value of **sysVar.transactionID** is the current transaction ID associated with the initial program in the current transaction.

If you use a segmented **converse** statement in a program started by a **transfer to program** statement from another program (for example, A uses a **transfer to program** statement to transfer control to B, which issues the **converse** statement), then the default transaction ID starts the original program (A) again on the input from program B’s **converse**. The generated program A reads the transaction status record from the work database, determines that B was the program that issued the **converse**, and transfers to B to continue processing. This logic is generated into the program for you.

You can bypass the overhead of restarting the original program through the following actions:

- Define a unique transaction ID for each segmented program started by a **transfer to program** statement.

- Move the transaction ID that corresponds to a program into **sysVar.transactionID** before running the first **converse** statement in that program.

Each transaction ID you use must be defined to IMS or CICS as being associated with its corresponding program.

The following two diagrams show the difference in program flow when you use a **transfer to program** statement with default transaction IDs, as opposed to setting the **sysVar.transactionID** variable.

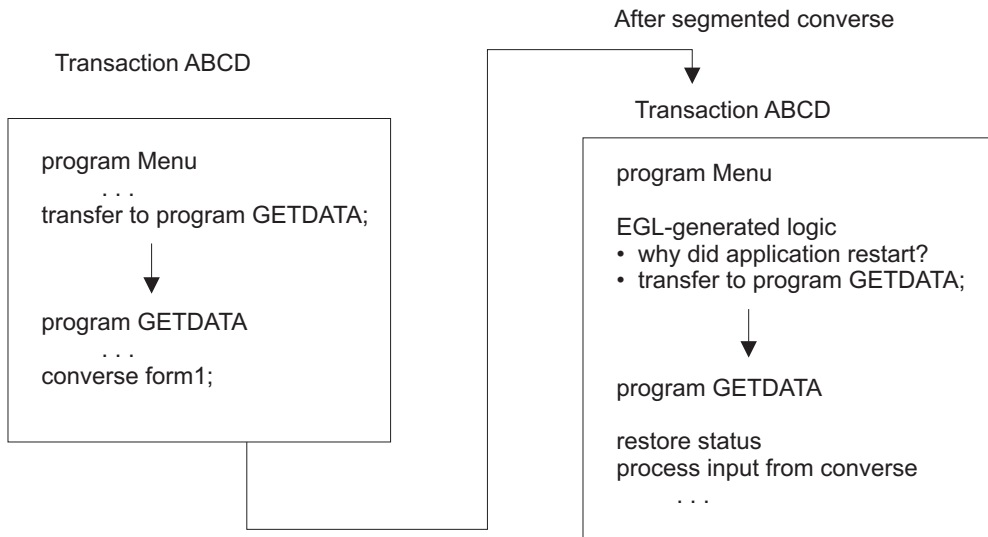


Figure 4. Example transfer to program using default transaction IDs

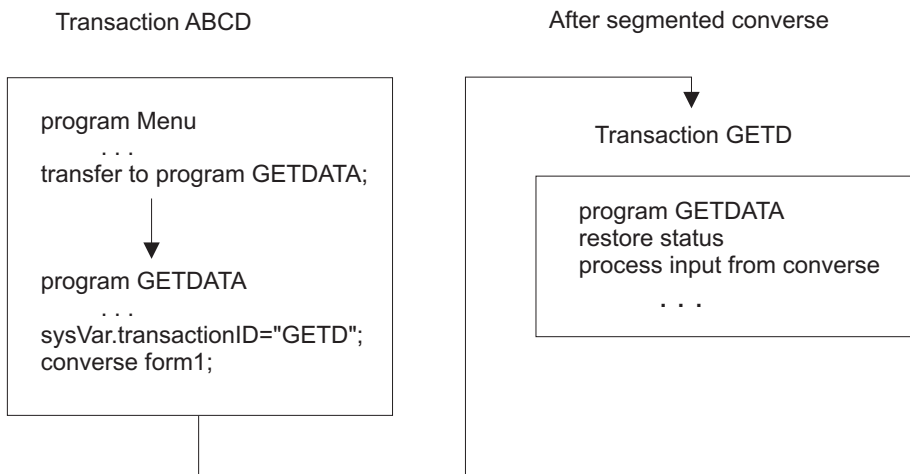


Figure 5. Example transfer to program using **sysVar.transactionID**

For CICS, if you use default transaction IDs, you need one RDO TRANSACTION entry to associate transaction ABCD with the menu program. If you call **transfer to program** and then set **sysVar.transactionID** to "GETD", you need two TRANSACTION entries, one to associate transaction ABCD with the menu program and one to associate transaction GETD with program GETDATA.

For IMS, if you use default transaction IDs, you need one pair of APPLCTN and TRANSACT macros to associate transaction ABCD with the PSB for the menu program. If you call **transfer to program** and then `setsysVar.transactionID` to "GETD", you need two pairs of APPLCTN and TRANSACT macros, one pair to associate transaction ABCD with the PSB for the menu program and one pair to associate transaction GETD with the PSB for program GETDATA.

Related information

"Program design considerations" on page 6

You should consider a number of factors when designing segmented programs.

"Using a show statement with inputForm"

Using a show statement with inputForm

With the **show** statement, a form name is required; a record name is optional. The generated program displays the form and identifies the next transaction to the CICS or IMS environment. The new transaction is scheduled when input is received from the program user.

When you use a **show** statement, you must specify the same form in the **inputForm** property for the transferred-to program. The **inputForm** property contains the name of the form that provides input to the program before processing begins. The transferred-to program begins by reading the same form displayed using the **show** statement in the initial program. You can use the **inputForm** property and **show** statement to create an IMS deferred program switch or a RETURN TRANSID for CICS.

Note: For IMS/VS, when you use a **show** statement and the **inputForm** property, the two programs must share the same FormGroup. For other environments, the form can be in different FormGroups, but it must be the same form.

When you specify the **inputForm** property for a program, the processing that occurs when that program is started varies:

- If a form is not received, the generated program automatically displays the form that was specified in the **inputForm** property.
- If a form is received, the generated program automatically performs any validation edits that are required before beginning the normal processing logic.

When you transfer program control using a **show** statement, you control the amount of data saved during user think time and the location where it is saved:

- You can specify a record when you define the **show** statement in addition to the form.

For the IMS/VS environment, Rational COBOL Runtime automatically saves the record in one of the following places:

- In the Scratchpad Area (SPA) for conversational processing where the **spaSize** build descriptor option is set greater than 0 and the **spaADF** build descriptor option is set to "NO".
- In the work database for nonconversational processing where the **spaSize** build descriptor option is set to 0 or for conversational processing when the **spaADF** build descriptor option is set to "YES".

For the CICS environment, Rational COBOL Runtime automatically saves the record in the COMMAREA.

- You can put additional data on the form by setting the **intensity** form field property to **invisible**. This allows the data to be available to the transferred-to

program when it reads the **inputForm**, but keeps the user from seeing the data when the form is displayed. A copy of the form is saved in the work database so the data can be displayed again if the user requests a help form.

For the IMS/VS environment, if you want to avoid saving a copy of the form in the work database, you must do the following:

- Set the **modified** property to YES for all variable fields on the form. You can do this when you define the default attributes for the fields on the form or with the **set** statement.
- Ensure that all the other properties are set to their defined values before the **show** statement.
- You can save data in a database using EGL **add** or **replace** statements and then restore the data in the transferred-to program.
- You can also use a **show** statement to transfer to the same program (program A can transfer to program A by using a **show** statement).
-

Related information

“Accessing multiple DB2 plans in z/OS CICS”

“Accessing multiple DB2 plans in IMS” on page 14

Accessing multiple DB2 plans in z/OS CICS

When creating a system of programs that access DB2 tables, you might not want to bind all the database request modules (DBRMs) for each program into one DB2 plan. For security and maintenance reasons, you might want to access several DB2 plans in a system of programs. This section discusses three of the possible methods for accessing multiple DB2 plans in CICS for z/OS. The first two methods describe how to change the transaction ID. The third method uses the DB2 Dynamic Plan Selection function.

If you have associated the DB2 plan name and transaction ID in an RDO DB2ENTRY or DB2TRAN definition, you can change the DB2 plan name by changing the transaction ID. For more information on the RDO DB2ENTRY and DB2TRAN definitions, refer to the appropriate installation or administration manual for your version of DB2.

Accessing DB2 plans using `sysVar.transactionID`

EGL system variable `sysVar.transactionID` enables you to dynamically change the segmented transaction ID. When running a segmented program, the value in `sysVar.transactionID` is used as the transaction ID to start the program again immediately after every **converse** statement. A simple example of using `sysVar.transactionID` to dynamically change the transaction ID is shown in the following diagram, where the initial value of the transaction ID is "Menu".

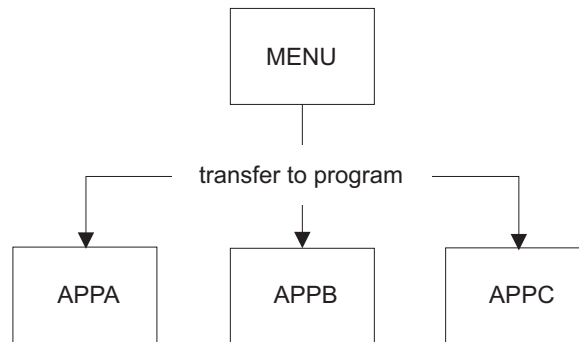


Figure 6. Example of using `sysVar.transactionID`

Table 1. Changes to transaction ID

Application:	APPA	APPB	APPC
Transaction ID (after converse):	AAAA	BBBB	CCCC
DB2 Plan:	PLANA	PLANB	PLANC

In this example, the menu program converses a menu form with three options. When the user selects an option, the menu program issues a **transfer to program** statement to the corresponding task-oriented program. Each of the three task-oriented programs moves a transaction ID to `sysVar.transactionID`, converses a form, and retrieves a row from a DB2 table. For example, the following logic could be used in each of the task-oriented programs:

```

...
sysVar.transactionID="AAAA"; // Set new transaction ID
converseInfoform();          // Converse information form to user
readDB2Record();             // Retrieve data from a DB2 table
...

```

There are no SQL statements in the menu program or, prior to the **converse** statement, in the task-oriented programs.

Each task-oriented program's DBRM is bound into a unique DB2 plan and associated with a unique transaction ID in the RDO DB2ENTRY or DB2TRAN definition. This is the transaction ID that is moved to `sysVar.transactionID` before the **converse** statement. After the **converse** statement, a new transaction starts with the plan associated with the new transaction ID.

The transaction IDs AAAA, BBBB, and CCCC are associated in the RDO DB2ENTRY definition with DB2 plans PLANA, PLANB, and PLANC, respectively.

This method of association can be used only in segmented programs. However, for CICS programs, you can access different DB2 plans with a **transfer to transaction** statement or by dynamically selecting a DB2 plan.

Accessing DB2 plans with a transfer to transaction statement

This method is useful if you transfer control between programs using a **transfer to transaction** statement. The transaction ID is changed when you transfer from one program to another using a **transfer to transaction** statement. This gives you access to a new DB2 plan if you associated the new transaction ID with a different DB2 plan in the RDO DB2ENTRY definition.

Dynamically selecting a DB2 plan

This method uses DB2 dynamic plan selection, which provides the ability to dynamically select a DB2 plan name for a z/OS CICS transaction. DB2 dynamic plan selection provides you with the option of defining a PLANEXITNAME in the RDO DB2ENTRY definition instead of a DB2 plan name. The exit program selects a DB2 plan for the z/OS CICS transaction. With this function, you can associate several plans with one transaction ID. For more information about DB2 dynamic plan selection, refer to your DB2 system documentation.

The first SQL statement in a logical unit of work (LUW) starts the exit program.

Related information

“Accessing multiple DB2 plans in IMS”

Accessing multiple DB2 plans in IMS

All the DBRMs that run together using a single IMS PSB must be bound together in a single DB2 plan. The following programs run together using a single IMS PSB:

- A main program and all programs it calls.
- A program and all programs that it transfers to using a **transfer to program** statement.

You can change DB2 plans with the following techniques:

- Transfer to a new program using a **transfer to program** statement, change the **sysVar.transactionID** system variable to a new transaction code, and then do a **converse** before using any SQL I/O options. For more information on this technique, see “Accessing DB2 plans using sysVar.transactionID” on page 12.
- Transfer control to a new transaction using a **transfer to transaction** statement. Different transactions can use different PSBs and can, therefore, have different DB2 plans.

Related information

“Accessing multiple DB2 plans in z/OS CICS” on page 12

Error processing for segmented programs

The test facility issues warning messages if it detects the potential for error. For example, warning messages are issued if a record is currently being updated, a **converse** is encountered, and the **converseVar.segmentedMode** system variable equals 1.

EGL validation and generation performs the following actions:

- Prevents you from referencing the **converseVar.segmentedMode** system variable unless you are running in VAGen compatibility mode.
- Prevents programs from being generated for IMS/VS if the **segmented** property is explicitly set to “NO”.

Related information

“Developing segmented programs in EGL,” on page 1

Index

R

runtime messages

 customizing EGL system messages 1,
 2, 5, 6, 7, 9, 11, 12, 14