

Modularizing Your Synergy Code: The First Step to Distributed Computing

Printed: July 2007

First printed October 1997. Revised September 2002.

The information contained in this document is subject to change without notice and should not be construed as a commitment by Synergex. Synergex assumes no responsibility for any errors that may appear in this document.

Synergex, Synergy, Synergy/DE and all Synergy/DE product names are trademarks of Synergex. Java and all Java-based trademarks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. ActiveX is a trademark of Microsoft Corporation. CORBA is a trademark of the Object Management Group.

© 1997, 1998, 2002, 2003, 2007 by Synergex

DCN WP-01-0005

Synergex
2330 Gold Meadow Way
Gold River, CA 95670 USA

<http://www.synergex.com>

phone 916.635.7300

fax 916.635.6549

Contents

| | |
|---|----------|
| The Distributed Computing Trend..... | 1 |
| Basic Techniques for Writing Modular Code | 2 |
| Strategies for Modularizing Synergy Code | 6 |
| Getting Started..... | 6 |
| Separating the User Interface from Application Logic | 6 |
| Separating Data Access from Application Logic | 7 |
| Using ELBs or Shared Images..... | 7 |
| Specific Coding Suggestions for Synergy Language..... | 7 |
| Common and Global Data | 7 |
| Using arguments instead of shared data | 8 |
| Using .INCLUDE files to encapsulate data definitions | 9 |
| Using a ^VAL Function to Return a Status Value | 10 |
| Alternative Approach: Wrapping a Routine | 11 |
| GOTOs and Error Handling | 13 |
| Error trapping..... | 13 |
| Expected errors in I/O statements | 14 |
| Expected errors in non-I/O statements..... | 15 |
| Unexpected errors..... | 16 |
| Calls to Labels..... | 17 |
| Applying these ideas to UI Toolkit..... | 18 |
| Re-entrant Routines | 19 |
| File I/O | 19 |
| General practices for encapsulating data file access..... | 20 |
| Handling user interface elements | 21 |
| Getting context information without passing it as arguments | 21 |
| Problems to consider in encapsulating data file access..... | 22 |
| Concluding Remarks | 23 |

Modularizing Your Synergy Code: The First Step to Distributed Computing

In recent years we've seen exciting advancements in technology accompanied by promises of a distributed, Web-enabled world. Distribution standards like DCOM (Distributed Component Object Model) and CORBA (Common Object Request Broker Architecture) have advanced to the point that they are finally delivering usable tools for building distributed multi-tier applications and for supporting technologies in deployment. The World Wide Web has contributed to the distributed computing trend by providing the ultimate thin-client distribution model, in which all user interface code operates on the client and all business logic and database access operates on the server.

These advancements are exciting for new development, but what do they mean for legacy systems? Current technologies require well-modularized, encapsulated code that can be distributed independently in a software system. In fact, modularization of code is the single most critical step towards distributed computing. For legacy systems that were not designed in a modular manner, however, it is difficult to take advantage of these new technologies.

This paper prepares Synergy™ developers for current and future trends in software development by clarifying what modular code is, describing its advantages, suggesting tactics for getting started, and providing solutions for issues specific to Synergy Language.

The Distributed Computing Trend

Many legacy systems are large, monolithic applications that handle all aspects of computing: they are responsible for the user interface, the database access, and all the business logic. Today, the trend is towards creating small modules or components that handle specific tasks required by an application. These modules may be distributed across multiple machines in a way that minimizes network traffic and optimizes overall system performance.

In distributed computing, processing is divided into essentially three layers: user interface processing is handled on the user's machine, database processing is handled on the machine (or machines) where the data resides, and business logic processing occurs on any machine on the network that can do the required processing. Properly modularized code—in which the user interface is not hard-wired to the business or database logic—enables Synergy developers to take advantage of a distributed processing environment. Developers can access required functionality from various applications and through various user interfaces.

Most developers are already familiar with procedural development techniques. Developers who practice good proceduralization techniques are already doing part of the work required to modularize their code. Procedural techniques encapsulate frequently-used functionality into utility-like routines, which are then re-used. This makes the code easier to maintain than code that is replicated throughout an application, and re-using pieces of functionality introduces another level of quality control into the development process.

Modularization is different from proceduralization in that it requires the code to be an isolated functional unit with a well-defined, published interface (i.e., an argument list). Modular code is like a black box: you cannot see the code or data inside. The published interface tells you how to use the module, and all functionality is provided through that interface. In contrast, procedural code is just a grouping of functionality that can be called

from elsewhere in the system; it does not have to conform to a published interface and its functionality can be affected by changes to shared data. The use of a published interface gives modular code several advantages over procedural code.

Because procedural code and data are exposed and visible to developers, they may use the side effects of a particular routine, or may use the whole procedure in a way that was never intended. Then, when the code is changed, it can cause problems throughout the application. Modular code inhibits this type of misuse because the inner workings of the code are hidden from view, and the functionality can be used only through the published interface in the intended manner. As with any type of coding, strict adherence to standards is essential in producing high-quality, bug-free code. With modular code, adherence becomes routine because the modules can be used only in accordance with the published interface.

Using modular code also makes your applications easier to maintain and extend, and decreases development time. Maintenance becomes less expensive because code is simpler and has fewer side effects. Functionality that is dependent on a routine being implemented in a particular way—as can be the case with procedural code—can cause dramatic and unexpected failures whenever that implementation is modified. The implementation of a piece of modular code, however, can be altered extensively without affecting the rest of the system because all communication with the code is accomplished through the interface. You need to consider the system-wide effects of a change only when the number or type of arguments defined in the interface are changed. Even in this case, the software system is easier to update because calls to the affected subroutines are well defined and you know exactly what they look like.

The time needed to develop new applications and extend existing applications decreases as code development becomes more efficient. Programmers do not need to fully understand the implementation details of a module as long as they know what it does and how to use the interface. This way, programmers can focus on the work at hand instead of puzzling out someone else's code.

Finally, applications built with encapsulated business logic are easier to componentize, ultimately enabling Synergy code to be integrated into CORBA and DCOM/ActiveX distributed systems.

Basic Techniques for Writing Modular Code

Synergy developers can write Synergy Language routines that take advantage of the latest technologies for distributed components and Web applications by adhering to a modular approach. But what exactly is a modular approach?

A modular design reduces complexity and eases maintenance. A modular program should contain “coherent” routines, “loose coupling” between routines, and a “clearly defined interface” to handle interaction between routines.

- ▶ A “coherent” routine performs a single, clearly defined task.
- ▶ “Loose coupling” between routines indicates that each routine can function as an independent building block, performing its task as required without depending on other routines. While routines can be combined in ways that allow complex functionality, they do not require any other routine to behave in a certain way or to set up certain variables.

- ▶ Interacting via a “clearly defined interface” requires that all the inputs and outputs of a routine be identified in its argument list, rather than as global or common variables. When designing modular code, the rule to remember is: For a routine to access external data, the data must be passed to the routine as an argument. To those using the routine, this rule might be stated as: Thou shalt not make any assumptions about the innards of a routine, nor shalt thou care about the implementation. Just pass the arguments defined in its interface.

Just as programmers sometimes make trade-offs between performance and space requirements, these three principles sometimes require trade-offs within your application. Although up-front design time can be longer for a modular system, striving to adhere to these principles will make your code more maintainable, extendable, and reusable. As time goes by, this will pay off in terms of decreased development cycles and improved quality.

The following twelve “rules” should help you in your efforts to create a more modular program design.

1. **Make each routine perform a single, coherent task.**

It should be possible to describe the task of each routine in your system in a single sentence without using words like “and” and “then.” For example, a routine that

- ▶ reads a user name and password
- ▶ validates that the user has access to the system
- ▶ draws the main menu
- ▶ and then allows the user to select the first option

violates the coherence principle. This routine is drawing a login screen, validating the user, drawing the main menu, controlling access options, and perhaps a host of other tasks.

The exact place to draw the line on what constitutes a coherent routine depends on the complexity of the steps and whether you ever want to use them separately. A single routine that draws the login screen and validates the user may be coherent if you only do user validation once. But if you periodically validate that a user has the right to a particular task within your system, and you do that validation without drawing a login screen, then it is wise to “decouple” these activities by writing a separate routine for each purpose. A separate routine might also be useful if you ever want to use the user validation routine in another application with a different login screen. In general, when in doubt, pick the more modular approach, because it grants you the most flexibility in the long run.

Note: The coherence principle does not preclude the possibility of a control routine that accomplishes multiple related tasks by calling other routines that perform the actual tasks. This structure can be extended by creating control routines that call other control routines.

2. **Avoid close coupling between routines.**

Close coupling occurs when a routine cannot accomplish its assigned task without the assistance of other routines. It is most dangerous when the assistance occurs in the form of undocumented side effects.

3. Give each routine a meaningful name.

A name that describes the task performed by a routine significantly improves the readability and maintainability of programs that include the routine. It also improves the reusability of the routine because programmers can easily identify and remember it.

4. Keep routines short.

There are many standards regarding the length of routines. Some people advocate that routines should never be more than 10 lines long; some say they should fit on a single computer screen; some say they should fit on a single page of paper. The exact size is not critical. The purpose of the “keep it short” rule is to add a physical constraint that enforces the idea of coherence and improves the readability of your code.

Note: This rule does not apply to comments!

5. Minimize or eliminate global and common data.

Global and common data (shared data) requires careful management. It works well when programmers carefully keep track of *all* the routines that may affect the shared data pool and are very clear about all the possible paths through the code. This is how programmers make sure that they can rely on the contents of the shared data variable that they are using in a given routine.

While it can be simple and straightforward to design programs using shared data, they can be burdensome and inflexible to maintain. Any programmer considering a change in logic has to consider whether that change affects a shared variable and goes on to create ripples throughout the system. One goal of modular design is to ensure that changes in a local routine have only local impact. This significantly improves the maintainability of the code because the programmer making a change needs to study only the routines related to that change, not the entire system.

Use of shared variables severely restricts the reusability of a piece of code, even when they are in an executable library (ELB). If the code is used in an environment other than the program it was written for, it will no longer have all its supporting variables in place, and it will not run correctly.

Note: This caution about using shared data also applies to static data allocated within a given routine. Static data is handled as global data and can introduce the same mysterious side effects.

6. Pass all data needed by the routine via arguments, and return output data via arguments or the result.

Passing all data through the interface defined by the routine definition allows a clear picture of what variables are affected by the routine, and whether any given change will affect a particular routine. It also allows the routine to be used in a library of routines that can be accessed by multiple applications.

7. Simplify the argument list by grouping related arguments into records.

The most common objection to passing all data through argument lists is that the lists can become quite long. In addition, if you change a routine’s functionality, your argument list may also require changes, and then all the code that uses that routine must be adjusted and recompiled.

Both of these problems can be solved by grouping related arguments into records and passing records. Commonly used data can easily be handled by creating a **current_environment** record to store all the fields currently stored globally. Using record definitions this way improves both the maintainability and readability of the code.

8. Make local variables local.

If a variable is needed only for the task that a particular routine accomplishes, it should be stored locally. This decreases the possibility that other programmers will rely on the side effects of your program.

9. Document assumed pre- and post-conditions in the header comment block.

When you know things about the values that are going to occur in your program domain, and you know things about the order in which routines are executed in your program, it is not uncommon to make assumptions about the state of the system at the time your routine is called. For example, you might write a read routine that assumes the files to be read are already open because you know the files are opened by the system initialization routines, and that your routine will never be called if those routines fail.

These assumptions do not cause problems until somebody changes the system or tries to use your routine elsewhere. Clearly documenting your assumptions identifies the problems that require attention during any reorganization of the system.

10. Validate that data passed in the actual arguments meets the assumed pre- and post-conditions.

Even in cases where it is reasonable to make assumptions about the state of the system, it is good programming practice to confirm that your assumptions are valid and to return an error if they are not. For example, if you are counting on the fact that all numbers received by your routine are positive, add a check to confirm this before using a number.

Note that adding such validations can sometimes reduce the coherence of your code (i.e., you validate and then act). However, the trade-off is increased robustness of the system and decreased coupling between routines. You can maintain a reasonable level of coherence by creating validation routines that perform common validations, and then calling those routines whenever a validation is needed.

11. Return status information as well as result information.

In many systems, the basic functionality is well-modularized but the exceptions (i.e., error conditions) are handled in-line. In other words, if you get an error reading a file, you might display an error message to the screen right there in your routine. This can cause subtle difficulties as you begin to separate the user interface from the application logic. As the system becomes more and more modular, you may not know where or when a given routine is being called. Consequently, it is best to return status information (i.e., success, failure, error numbers or messages) via the return result or the argument list. This allows the caller to handle the error as best fits the situation when the routine is actually called.

12. Avoid explicit GOTOs and localize implicit GOTOs.

GOTOs send program control to remote corners of the source code with no assurance that it will return. Instead of GOTOs, use calls to internal subroutines. Maintaining a modular structure within programs will aid maintenance and make code more accessible to all developers.

Strategies for Modularizing Synergy Code

Up to now, we've been discussing some general concepts of modularization. The next step is to apply these concepts to applications written in Synergy Language. This section presents general strategies to help you approach the task of modularizing your code. The next section, "[Specific Coding Suggestions for Synergy Language](#)" on page 7, offers suggestions on how to handle specific Synergy Language issues, such as error trapping and GOTOs.

Getting Started

If you are designing an application from the ground up, it makes sense to think in terms of completely modularizing your whole system right from the start. However, in many cases, the goal is to improve an existing application. This makes things a little more complex, and the task may seem overwhelming.

The first step is to think about where you most need the benefits of modularization. What pieces of functionality do you want to make available on an intranet or the Web? What user interface components require the flexibility of an ActiveX control? What routines could you centralize into a utilities library that could be used by many or all of your applications? Do you want to have multiple user interfaces—perhaps a cell-based user interface and a Web or GUI interface? Are you planning to support both Synergy ISAM files and relational databases?

Once you've identified the areas where modularization is needed, you can start redesigning the required functionality with your objectives in mind.

- ▶ If supporting multiple user interfaces is your priority, the first step is separating the user interface from the application logic.
- ▶ If you want to support multiple databases, the first step is separating the file access from other application logic.
- ▶ If you plan to expose your order tracking system to your customers on the Internet, you will first need to modularize the system's application logic so that it can be called remotely from the user's web browser. Another option is to write a few modular routines that perform the functions you want customers to have access to, and then integrate them into your order tracking system at a later date.

As with any other significant code modification, you can handle the move to modularity in chunks, slowly moving your system to a more and more modular state.

Separating the User Interface from Application Logic

When your user interface and application logic are separate, there are some routines that control what the user sees (e.g., input windows, error messages, prompts) and separate routines that handle data access, calculations, validations, report generation, etc. When you need the user interface to display the result of a piece of application logic, the user interface logic calls a function that performs the required task and then displays the result.

UI Toolkit, which supports event-style programming, effectively separates much of the user interface from the data management. However, Toolkit event loops are candidates for revision if you are planning to support multiple user interfaces. When you process user events using arrive/leave methods or break processing, be sure to handle those events with a call to a function in an ELB. This function must return a result that your user interface logic processes.

Error messages are an often-overlooked part of the user interface. Your routines should return a success or failure status, and the user interface code should handle the actual generation of the error message on the screen. See [“Error trapping” on page 13](#) for more information.

Separating Data Access from Application Logic

Separating the database from the application logic involves writing routines to handle all file I/O. This is probably the most commonly modularized area in existing applications because it hides the details of file organization, data access, and so on from the user of the routine. Since file structures change frequently, the benefits of doing this have been obvious for a long time. Again, be sure to check carefully for error messages, data validations, or other functionality that belongs in other layers of the distributed client server system or Web application model. See [“File I/O” on page 19](#) for more information.

Using ELBs or Shared Images

Executable libraries (ELBs) and shared images (on OpenVMS) provide a means of storing related routines in a common library that can be called by multiple Synergy programs. As you move to distributed computing, it will be possible to use the same routines with a variety of applications and user interfaces.

The ideal approach to modularizing your system is to create a set of routines with defined interfaces (formal argument lists) and then group related routines into ELBs. For example, you might group your utility routines (e.g., string handling, date calculations) into UTILS.ELB, your order entry application logic into OELOGIC.ELB, and your order entry data access into OEDB.ELB.

These routines can be called by any application that can make use of them, particularly if you do not maintain global or common data inside the ELBs. With remote execution of ELBs, you can position each ELB where the application that uses it is located. The ELBs can be opened at run-time as they are needed.

Specific Coding Suggestions for Synergy Language

This section contains suggestions for dealing with specific issues in Synergy Language in a modular way. These suggestions are indicative of the changes necessary to modularize your applications. Although there will be situations particular to your applications, the approach is universal: information is passed as arguments, routines perform single functions, and code is separated into user interface, database, and business logic.

Common and Global Data

The COMMON and GLOBAL (global data section) statements permit access to data objects outside a given routine. They allow routines to share records and variables without having to pass the data as arguments. In order for a routine to take advantage of shared data, it must be linked with the routine in which the shared data is defined.

The use of shared data severely restricts the reusability of code because it causes a routine to be dependent on an external routine. In addition, maintaining code that uses shared data can be troublesome because when a routine containing shared data is modified, the shared

data may change too, which can affect any other routine that links to that data. Furthermore, a programmer using the linking routine needs to understand the inner workings of every routine that references the shared data in order to use it correctly.

For purposes of modularity, each module needs to be a separate, stand-alone entity that can be called by any other Synergy routine, as well as by non-Synergy routines. All communication with a module will be through a defined interface. While at first glance this might seem restrictive, in reality it provides tremendous flexibility that enables transparent replacement of components as technology advances.

Using arguments instead of shared data

The solution to the problem caused by shared data is simple: remove all COMMONs and GLOBALs from routines that you want to expose to the outside world. In their place, specify arguments that contain the data being passed and returned.

This solution requires three steps:

1. Declare the data as records in the subroutine.
2. In the startup section of the routine assign each argument to its corresponding record area; it may be advantageous to use the GROUP statement to define the arguments within the record.
3. In the closing section of the routine assign any data objects that are returned to the corresponding arguments.

The sample code below illustrates this approach, using group argumentation. The original code is on the left; the altered code is on the right. The changes are bolded.

| | |
|---|---|
| <pre>.main my_main global common qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 record alpha ,a20 mask ,a*,"\$\$\$,\$\$X.XX" .proc qty = 3 price = 1234.5 call compute_discount alpha = total, mask .end .subroutine compute_discount .define FACTOR, 1000 external common qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 .proc disc = qty * price / FACTOR disc = (100 - disc)/100 total = qty * price * disc xreturn .end</pre> | <pre>.main my_main record arguments group pricing ,a qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 endgroup record alpha ,a20 mask ,a*,"\$\$\$,\$\$X.XX" .proc qty = 3 price = 1234.5 xcall compute_discount(pricing) alpha = total, mask .end .subroutine compute_discount group pricing ,a qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 endgroup .define FACTOR, 1000 .proc disc = qty * price / FACTOR disc = (100 - disc)/100 total = qty * price * disc xreturn .end</pre> |
|---|---|

Using .INCLUDE files to encapsulate data definitions

The necessary code changes can be further simplified by judicious use of .INCLUDE files. Using the same code example, if the fields in the original record for the COMMON are defined in MYFILES:my_fields.rec and are .INCLUDEd in the source, do the following:

1. Edit MYFILES:my_fields.rec and enclose the fields in a GROUP statement.
2. Change the definition in the main routine from a GLOBAL COMMON to RECORD.
3. Delete the EXTERNAL COMMON line from the subroutine.
4. Use .INCLUDE "my_fields.rec" as the first subroutine argument.

Note: This inclusion can be done from a file or from the Repository.

This method allows the contents of the arguments themselves to be referenced as they were previously, and requires no change to the procedural code. Using .INCLUDE files can make maintenance easier too, because you make changes in only one place, instead of throughout the code. The code sample on [page 10](#) illustrates this method. Once again, the original code is on the left, with the altered code on the right.

| | |
|---|---|
| <pre>.main my_main global common qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 record alpha ,a20 mask ,a*, "\$\$\$,\$\$X.XX" .proc qty = 3 price = 1234.5 xcall compute_discount alpha = total, mask .end .subroutine compute_discount .define FACTOR, 1000 external common qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 .proc disc = qty * price / factor disc = (100 - disc)/100 total = qty * price * disc xreturn .end</pre> | <pre><This is the file my_fields.rec> group pricing ,a qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 endgroup <This is the Synergy program file> .main my_main record .include "my_fields.rec" record alpha ,a20 mask ,a*, "\$\$\$,\$\$X.XX" .proc qty = 3 price = 1234.5 xcall compute_discount(pricing) alpha = total, mask .end .subroutine compute_discount .include "my_fields.rec" .define FACTOR, 1000 .proc disc = qty * price / factor disc = (100 - disc)/100 total = qty * price * disc xreturn .end</pre> |
|---|---|

Using a ^VAL Function to Return a Status Value

As mentioned in rule 11 (see [page 5](#)), it is important that your routines return a status value (in addition to any results) to indicate whether they completed successfully.

When writing new routines, you can write them to return a status in the argument list or as the return value of the function or ^VAL function. Existing Synergy Language subroutines can be modified to return a status by converting them to ^VAL functions and then using the ^VAL return value to return the status. The advantage to this method is that the argument list need not be altered to accomodate a status value. This means that if you make calls to the routine elsewhere in your code, those calls do not need to be modified.

The example below shows how to convert an external subroutine that does not return a status to a ^VAL function with a status return value. We have added code to the routine so that it returns a meaningful status value, but the argument list is not altered. Remember that you must declare the function as a ^VAL function in your code before calling it, or use the undefined functions compiler option (-X).

| Subroutine | ^VAL Function |
|---|---|
| <pre>.subroutine divide a_result ,n a_arg1 ,n a_arg2 ,n .proc a_result = a_arg1 / a_arg2 xreturn .end</pre> | <pre>.function divide, ^val a_result ,n a_arg1 ,n a_arg2 ,n record status ,n .proc clear status, a_result if (a_arg2 .ne. 0) then a_result = a_arg1 / a_arg2 else status = -1 freturn status .end</pre> |

Although the resulting routine is a ^VAL function, any existing code that calls it as a subroutine, for example

```
xcall divide(result, arg1, arg2)
```

is still valid—the freturn value will simply be ignored.

Alternative Approach: Wrapping a Routine

When you create a new routine with a defined interface, old code that calls this routine must be changed to conform to the interface. Updating all calls to the routine throughout the program will generally improve the modularity of your program, bringing with it increased reusability, maintainability, and extendibility. There may be times, however, when you need to expose a routine with a defined interface but are unable to update all the calls made to it. In this case, creating a “wrapper” routine that maintains the old interface can be a useful interim approach.

The wrapper encapsulates the routine that is to be exposed for external use. The benefit of this approach is that it does not require any changes to existing code, and allows routines to be exposed as they are required.

Note: This method provides only an interim solution to situations in which you have a multitude of calls to an existing routine throughout an application, and insufficient time to perform the design and multiple edits required to alter the data interface. It is an approach that involves some duplication of code, so all the problems and cautions associated with maintaining duplicate code apply. We recommend that as time allows, the calls be changed to use the new interface.

The concept is simple: For each routine that is to be exposed, extract the code into another routine, which is called with the global data passed as arguments. The original routine, which references the global data, now calls the new routine, and passes the data as arguments. The following example shows how this can be accomplished.

| | |
|--|--|
| <pre> .main my_main global common qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 record alpha ,a20 mask ,a*,"\$\$\$\$,\$\$X.XX" .proc qty = 3 price = 1234.5 xcall compute_discount alpha = total, mask .end .subroutine compute_discount .define FACTOR, 1000 external common qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 .proc disc = qty * price / factor disc = (100 - disc)/100 total = qty * price * disc xreturn .end </pre> | <pre> .main my_main global common qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 record alpha ,a20 mask ,a*,"\$\$\$\$,\$\$X.XX" .proc qty = 3 price = 1234.5 xcall compute_discount alpha = total, mask .end .subroutine compute_discount external common qty ,i4 disc ,d6.4 price ,d10.2 total ,d14.2 .proc xcall di_compute_discount & (qty,disc,price,total) xreturn .end .subroutine di_compute_discount qty ,n disc ,n price ,n total ,n .define FACTOR, 1000 .proc disc = qty * price / factor disc = (100 - disc)/100 total = qty * price * disc xreturn .end </pre> |
|--|--|

The data-independent subroutine named `DI_COMPUTE_DISCOUNT` can now be referenced by external sources without using any shared data. Synergy Language routines that use the shared data do not need to be changed to use the new subroutine because there is a wrapper, which actually contains the logic, around the subroutine. Calls to `COMPUTE_DISCOUNT` still reference the shared data as before, but the logic has been isolated into a subroutine that uses argumentation, and is therefore available to external calling routines.

GOTOs and Error Handling

The GOTO and ONERROR statements, once mainstays of many DBL applications, send program control to remote corners of the source file with no assurance that it will return. While there is nothing inherently bad about GOTO and ONERROR, their use makes it far easier to create fragmented, disorganized code, making these statements unsuitable for modular code. Without a stringently applied set of rules, efforts to track such code waste time at best, and often end with the programmer giving up in frustration.

These problems are solved with a simple rule: Never send program control away from the current location without ensuring its return. In other words, no GOTOs, and all error trapping must be directly associated with the statement that generates the error. Simple enough, but how do you accomplish this?

Today's most efficient applications replace yesterday's top-down, linear logic with a circular, event-driven structure. Consider the following code fragments.

| With GOTOs | Without GOTOs |
|--|--|
| <pre>call display_menu call get_choice if (choice.eq."sel1") goto sel1 if (choice.eq."sel2") goto sel2 if (choice.eq."sel3") goto sel3 if (choice.eq."sel4") goto sel4 if (choice.eq."exit") goto exit</pre> | <pre>done = %false do begin call display_menu call get_choice using choice select ("sel1"), call sel1 ("sel2"), call sel2 ("sel3"), call sel3 ("sel4"), call sel4 ("exit"), done = %true endusing end until (done)</pre> |

Not only is the sample on the left far less efficient (each IF test is executed until a match is found), it is also far more difficult to follow. Just where does each label go when it's finished? To learn the ultimate fate of each GOTO, you need to track down the associated label, and probably several more GOTOs in the process. In the event-driven version on the right, a tight loop (incorporating the more efficient USING statement) executes until the user makes the "exit" selection. Anybody looking at this section of code can presume—without checking the associated internal subroutine—that control must return to this loop.

Note: Because subroutines may exit or chain, this presumption may not be valid. Your coding standards can address conditions under which it is acceptable for a subroutine to transfer control.

Error trapping

But what about error trapping? After all, error handling with Synergy is nothing more than an implicit GOTO, regardless of the approach taken. This limitation makes it all the more important that Synergy errors be managed in a consistent, incident-specific manner. In handling errors, care must be taken to comply with the "loose coupling" principle: do not allow an error to send program control to the far reaches of your source file. In addition, no confusion should exist as to how execution flows to any section of code.

A routine should contain its own error trapping: it can signal back to the world an error state, or process the error and continue, but in either case the logic to handle specific error conditions has to be encapsulated within the routine.

In handling errors, there are three areas to consider:

- ▶ expected errors in I/O statements
- ▶ expected errors in non-I/O statements
- ▶ unexpected errors

Expected errors occur by design and allow processing to continue once they are handled, such as an end of file, or when a user searches on something that is not in the database or enters the wrong type of data in a field. *Unexpected errors* are unanticipated actions on the part of the user or the program; these are errors for which you do not have an error message prepared or a standard method of handling.

In planning your error handling, you should determine the types of errors that a statement could generate, and handle as many of them as possible in a specific manner. Then you will also need to develop a generic way to deal with any unexpected errors.

You can start by eliminating ONERROR from all I/O error handling, replacing it with the I/O error list that can be associated with any I/O statement. And when an error is trapped, the code that handles it must have an exclusive association with the statement that generates it.

Using statements like:

```
onerror process_error
```

to handle anything but unexpected, potentially fatal errors is a maintenance nightmare because it is difficult to tell what application error occurred or what logic generated it.

Expected errors in I/O statements

Expected errors need to be handled as specifically as possible, without jumping to another area of the program. The sample code below demonstrates the “old” approach to error processing, and suggests a localized, error-specific approach. This example illustrates how to handle a “file not found” error; you would, of course, need to account for all the other expected errors that could occur, as well as develop a method of handling unexpected errors (see “[Unexpected errors](#)” on page 16).

| This: | Becomes: |
|--|--|
| <pre> open_it, clear trys onerror (18) make_file, (38) try_again open(dat_ch, i, "cusdat") offerror . . make_file, offerror open(dat_ch, o, "cusdat") call load_file goto continue_here try_again, offerror incr trys if(trys.ge.10) call drastic_measures goto open_it </pre> | <pre> begin clear trys open(dat_ch, i, "cusdat") & [\$ERR_FNF=make_file, & \$ERR_FINUSE=try_again] exit make_file, open(dat_ch, o, "cusdat") call load_file exit try_again, incr trys if (trys.le.10) then begin open(dat_ch,i, "custdat") & [\$ERR_FINUSE=try_again] exit end else call drastic_measures end end </pre> |

The sample on the left is the old “tried and true” approach to error handling: set the trap, execute the statement, send control to some other section of the program to handle the error, and finally GOTO someplace else to resume processing. And heaven help the programmer who forgets to turn off the trap! On the right the ONERROR has been replaced by an error-specific I/O error-list trap; the error is handled right there with the code that generates it.

Expected errors in non-I/O statements

If an expected error occurs in a non-I/O statement, you must use ONERROR. But that doesn’t mean that control needs to go elsewhere. The sample code below uses ONERROR to trap a bad digit error.

```

onerror($ERR_DIGIT), bad_digit
if (%FALSE)
  begin                               ; Only executes if bad digit error is processed
bad_digit,
  display(tt_ch, 7)                   ; Beep
  end
xcall getval("Enter numeric value", avar)
dvar = avar
offerror

```

There is no rule that requires an error trap to send control to a higher line number (lower in the source file). In the above example, an error means that the user must re-execute a previous action. Control is sent to a label that executes only if the specified error occurs (IF tests for truth); when the error is handled, the program flows right back to the original code. The program cannot advance until it receives valid input.

Unexpected errors

Of course, when an unexpected error does occur, your program must account for it and respond accordingly. When a subroutine cannot recover completely from an error, control should return to the calling routine with information on the error. This is the only circumstance that justifies a generic, nonspecific ONERROR statement. The example below demonstrates this approach.

```
.subroutine important_stuff
  a_var1      ,a
  a_var2      ,a
  a_err_flg   ,n      ; Returned true if unexpected error

record
  dvar        ,d1

.proc
  onerror unknown_error      ; Trap any unexpected error
  dvar = "a"                  ; Generate error
  if (%FALSE)
    begin                    ; Only if unexpected error is trapped
unknown_error,
      xcall fatal_err(a_err_flg) ; Log error
    end
  xreturn
.end

.subroutine fatal_err          ; Log fatal error
  a_err_flg   ,n
.define ERRLOG ,99            ; Error logging channel

record
  modnum      ,i4          ; Previous module in calling chain
  routine     ,a30        ; Routine that module was called from
  line        ,d5          ; Line in routine that module was called
from
  done        ,i4

.proc
  done = %false
  open(ERRLOG, o, "error.log") ; File in which to log error
  do                ; Until no more modules in calling
chain
  begin
    incr modnum          ; Previous module
    xcall modname(modnum, routine, line)
    if (.not.routine)
      done = %true      ; No more modules
    if (a_err_flg) then ; Was error already logged
      writes(ERRLOG, "Called from line "+%string(line)+
& " in routine "+%atrim(routine))
    else                ; Log error module and line number
      writes(ERRLOG, "Error "+%string(a_err_flg=%error(line))+
& " at line "+^a(line)+" in routine"+
& %atrim(routine))
    end
  until (done)
  close ERRLOG
  xreturn
.end
```

This example shows a subroutine that is called from elsewhere in the program. Among the variables passed is an error variable to be tested upon return. The subroutine handles its standard errors in the normal fashion, but when an unanticipated error occurs, control goes to the *unknown_error* label and regular processing is terminated immediately. The *unknown_error* label calls the `FATAL_ERR` subroutine, which was written to handle unexpected errors. This subroutine returns information about the error. The error variable is loaded with the error number, and the subroutine returns to the mainline routine. If the error variable returns non-zero, the routine exits.

Your implementation will likely be more robust than this example, but this is the approach to take. This example assumes that it is appropriate in your application to return the status in an error log file. However, if you are operating in a distributed environment, you might want to return a string of information that the user could see, or perhaps create a log file for the system administrator and return an error message to the user.

Calls to Labels

A call to label transfers control to a specific statement during program execution. It allows the programmer to make the code more readable and have an internal routine that the entire module can use. With a little planning, some internal routines can be turned into external routines that can be used by other modules in the application. How can you determine if an internal routine will make a useful external routine?

First, determine what the routine does and whether it would be useful in other parts of the current application or in other applications. If it performs more than one task, the next step is to isolate each task into a separate routine so that other modules or applications can use the new routines. File I/O, field validation, error processing, and formatting are good examples of the types of routines you might create. Keep the user interface separate from other functionality so that your new routines can be used by other applications.

Once you have identified the subroutines that you want to create, the next step is to define the interfaces. These new subroutines need to have access to all the same data that the internal routine required in the form of arguments. Scan the routine for shared variables, and then handle them in one of these ways:

- ▶ Move each shared variable that the routine uses into the formal argument list for the subroutine.
- ▶ Turn the shared variable into a local variable.
- ▶ Access the shared variable via a call to another subroutine that can furnish the data.

One of the objectives in creating subroutines is to create a simple and intuitive interface that can remain relatively constant over time. Breaking the internal routine into multiple subroutines, each performing a specific task, can help minimize the number of arguments required. Even so, the argument list for a subroutine can become quite long. To make passing arguments less cumbersome, you can combine them into records or groups. This is similar to the method we employed when dealing with shared data on [page 9](#): define the records or groups in `.INCLUDE` files and `.INCLUDE` them in the subroutines. For example, the open channels required for the module and their open status can be passed in one argument as a record. A control record that tells the routine what task to perform and the success status of the subroutine can be combined in another record. Fields associated with error processing can be passed as a group.

Using .INCLUDE files also simplifies the maintenance issues related to the interface. By .INCLUDIng all these interface definitions in the calling routines, changes to the interface can be easily incorporated by simply recompiling the calling routines. The code samples below show how these ideas can be implemented. In the example on the left, a subtotal and sales tax amount are calculated using calls to labels. The code on the right shows how this example could be made modular by using argument lists.

| | |
|--|--|
| <pre> global common price ,d6.2 total ,d10.2 ,0.0 tax_amt ,d6.2 tax_rate ,6.4 ,0.0775 .proc . . call get_sub_total call calc_tax . . get_sub_total do begin reads (TTCHN, price) total = total + price end until (price.eq.0.0) calc_tax tax_amt = total * tax_rate . . .end </pre> | <pre> global common price ,d6.2 total ,d10.2 ,0.0 tax_amt ,d6.2 tax_rate ,6.4 ,0.0775 .proc . . xcall get_sub_total (total) xcall calc_tax (total, tax_rate, tax_amt) . . .end .subroutine get_sub_total total ,n record price ,d6.2 .proc do begin reads (TTCHN, price) total = total + price end until (price.eq.0.0) .end .subroutine calc_tax total ,n tax_rate ,n tax_amt ,n .proc tax_amt = total * tax_rate .end </pre> |
|--|--|

Applying these ideas to UI Toolkit

One way to maximize the value of isolating business logic into routines is to reuse them. Componentized routines can be called from multiple user interfaces, multiple modules, and even multiple applications. In a UI Toolkit application, you can call these routines from either a break processing code block or a leave method. In both cases, you would handle anything that was directly related to the user interface in the code that surrounds your componentized subroutine call.

The code skeleton for both of these approaches would be:

- ▶ any user interface processing that should precede the external subroutine call, such as initializing the parameters, adjusting the current field, etc.
- ▶ xcall the external subroutine
- ▶ any user interface processing that is required as a result of the external subroutine, such as putting the results of the external subroutine call into a user message or into an appropriate place on the screen

Re-entrant Routines

Re-entrant and recursive routines may require additional analysis in order to protect data related to individual invocations of the routines. Re-entrant routines must be modular. It is important that the re-entrant features of Synergy Language are used properly in order to ensure that re-entrant routines remain modular. Warnings against using shared and static data apply here as well. All information needed by a re-entrant routine should be passed as arguments, and local variables within the routine should be allocated on the stack and initialized with each invocation.

The following summation function shows an example of correct use of re-entrancy. This summation function sums a series of sequential numbers from any positive number n to 1,

e.g., $\sum_{x=1}^n x$ or $5+4+3+2+1$.

```
.function summation, ^VAL, REENTRANT
    value      ,n
    if (value.gt.1) then
        result = value + %summation(value-1)
    else
        if (value.eq.1) then
            result = value
        else
            result = 0      ;error condition
```

File I/O

All direct file I/O should be handled by a set of routines specifically for this purpose. For many systems, this is the first area to benefit from modular programming techniques. As system requirements change, there is generally a corresponding change in the data that must be stored and/or retrieved. Changing file formats and adding or deleting access keys can be hidden from the rest of the system by centralizing the actual work of creating an interface between the system and the files it uses into a few routines. Most mature applications already include a set of routines for *file_open()*, *file_close()*, *file_read()*, and *change_file_key()*. Depending on your application, it may also be advantageous to access particular fields of a file with *get()* and *set()* routines, thereby hiding all information about what fields exist in what files. This approach enables programmers to make changes to their applications fairly quickly when the underlying data files change.

With the advent of client/server architecture and distributed systems, it becomes more critical than ever to create data access routines to handle all data access. Doing so allows multiple applications to share related data, while making it easier to maintain the routines that handle the data by centralizing and encapsulating them.

It is also important to scrutinize old data access routines to ensure that they have no user interface components, such as prompts or error messages that are displayed to the end user. When using multiple user interfaces and similar interfaces written in multiple languages, it becomes critical to make sure that the information required for generating an error message is passed to the routines that are actually controlling the user interface.

General practices for encapsulating data file access

The first step in encapsulating your data file access is to identify the types of functionality that you need. For example, in some applications, you may want generic routines for *file_open()*, *file_close()*, and *record_read()*. In other applications, you might want a set of routines for *read_file_y()*, *read_file_x()*, *read_file_x_cache()*, etc. This decision will depend on how similarly you perform each type of processing. If you find yourself coding the same steps over and over again, you should write a generic routine. If you find yourself creating dozens of flags to make a generic routine perform special processing under special circumstances, you would probably be better off with a set of more specialized routines.

Once you have determined what functionality you want the routines to perform, the next step is to identify the information that should be passed back and forth to accomplish that result. The key to good encapsulation is to create a routine that can be run effectively as a stand-alone entity: just pass it what it needs to process and it will return the desired results. Generally, this means that you should pass the required input information and variables that can hold the required output via the actual argument list. (An alternative approach is discussed in [“Getting context information without passing it as arguments” on page 21.](#))

The information needed for opening a file includes the filename and the mode for opening (e.g., read-only, update). In your application, you might also need user name, number of times to retry before failure, and other information that is relevant to the way you handle file access. The subroutine needs to return the channel number that the file can be read on. The argument definition for a *file_open()* subroutine might look like this:

```
.subroutine file_open
  filename           ,a
  open_mode          ,n
  channel_num        ,n
  status_code        ,a
```

The *status_code* field returns success or failure information. Handling status this way allows the calling routine to determine how to handle the error, including how to handle it in the user interface.

The information required for reading a particular record includes the channel number that the file can be read on, the record that should be returned, information identifying the record that should be read (a key, a partial key, or perhaps a flag that indicates you are doing a sequential read), and any additional information relevant to the way your application handles reading a particular file. In keyed file reading, it is common practice to pass the key information in the record that will be returned. The argument definition for *read_myfile()* might look like this:

```
.subroutine read_myfile
  channel_num        ,n
  myfile_rec         ,a
  status_code        ,a
```

This looks fine at first glance, but problems emerge when you consider common file I/O issues like duplicate keys, processing a list of partial key matches, and sequentially accessing a file for information that is not accessible via a keyed search. As currently structured, our example does not have enough information to handle these more complex issues. One solution is to add flags that indicate what kind of action should be performed in these special cases, so that *read_myfile()* might look like this:

```
.subroutine read_myfile
  channel_num      ,n
  myfile_rec       ,a
  partial_key_flag ,n
  read_type_flag   ,n      ; EXACT,
                           ; KEY_START,
                           ; NEXT_REC

  db_err_no        ,n
  status_code      ,a
```

This approach can lead to complicated argument lists, and if you want to add new functionality, you may need to add another flag (and then make a corresponding change in every call to the subroutine). Some application programmers create a status record to return error information, and a “control” record to contain all the information—other than the actual data—that you need to pass. This allows you to reap the benefits of modularization without creating unwieldy argument lists. In addition, the maintainability of the code is greatly simplified, because the control, status, and data records are described in `.INCLUDE` files, which can be changed without changing the interface. When we add this concept to our example, the argument definition of *read_myfile()* looks like this:

```
.subroutine read_myfile
  myfile_rec      ,a
  control_rec     ,a
  status_rec      ,a
```

Handling user interface elements

If your data access routines will be used in a distributed system, it is best to replace all user interface elements in the routines with information passed through an argument list. For example, information that might have been collected from the user through a prompt or pick list may need to be passed as an argument or a field in a control record. Error codes can be returned rather than handled directly through the generation of an error message.

Getting context information without passing it as arguments

It is important to realize that argument lists are not always the best way for subroutines to get information about the context of your application. Argument lists should be used when the calling routine already has the information that needs to be passed or uses that information in some way.

In some cases, however, it may be more useful to create a specialized subroutine that can be called when needed to retrieve specific context information. You may want to use this approach when the information needed by the subroutine is basic or general information that is used frequently throughout the system, or when it is cumbersome to pass a lot of information around the system.

For example, let's say you have an order entry system that keeps a detailed context record somewhere that includes information like user name, current order number, currently selected item, last item, remaining credit limit, and so on. There's a lot of information in this record, so perhaps you don't want to pass it all over the system as an argument to every subroutine. This problem can be solved by creating subroutines like *get_remaining_credit_limit()* and *get_user_name()* to retrieve specific pieces of information when needed. This allows you to encapsulate the details related to how you store valuable context information, so that the details can be changed without breaking existing code.

Problems to consider in encapsulating data file access

Your encapsulated data access routines need to take into account all the same things that can go wrong with unencapsulated routines. In addition, since you can't predict all the ways someone else might wish to reuse your routines, you must be extra careful to validate any assumptions that you make and to return an error message when those assumptions are violated.

For example, in an encapsulated *record_write* routine, it may not be safe to assume that the last record read will be the same record that you need to overwrite. Similarly, using RFAs (record file addresses) as a means of accessing a given record has to be considered carefully. In order to ensure that you can write a record safely, it may be necessary to position the file pointer correctly in your write routine, rather than assume that the positioning was done correctly by other routines and has not been altered. Adding this logic to your routine makes it more robust; you can now use the routine safely without knowing the order in which other routines have been called. This is a good example of the benefits of encapsulation.

If you are using static RFAs, you may want to pass the RFA around with read and write routines to improve performance in a distributed environment. However, you should weigh the improved performance achieved with static RFAs against the loss of data space: if you frequently add and delete records from your database, the trade-off is probably not to your advantage.

Reading a file with duplicate keys, partial keys, or unique keys that may not be accessible until after you've already read the record (such as keys that include timestamps or dates) is a special challenge in the encapsulated environment. The calling routine needs to be able to keep calling until it gets the right record, or the caller may need to return an array of possible matches. These possibilities must be considered when constructing the data access routines.

If your application creates temporary files for things like sorting on an unindexed field, it is a good idea to create a subroutine that handles that process, and have it return an exhaustive set of error codes. This way, any routines that rely on the presence of a temporary file can propagate the error back up the calling chain in the case of failure. If two or more routines share a temporary file (i.e., one creates it and one reads it), you should include the name or channel of that file in the argument lists for the related routines, or create a function to supply it.

Concluding Remarks

Be aware that adding modularization to an existing application can be more difficult than designing a modular system in the first place. It is important to identify all the areas that need attention—the shared data that must be replaced, the functions that have been handled with a call to a label, the error messages that are now displayed on the screen—and work them through carefully. Be sure that you plan plenty of time for testing because you may not catch all the assumptions that you've violated with your reorganization just by reviewing the code.