# Overview

During this exercise you will refine the input processing code that you created in the previous exercise, removing some of the “anomalies” which are currently present.

# Resources

* [Synergy DBL Language Reference](https://www.synergex.com/docs/index.htm)
* [Synergy Best Practices - Coding Standards](http://jobfunc2.cu.net/Job%20Functions/Programmer/Programmer%20Handbook/Tims%20Best%20Practices%20-%20Standards/Synergy%20Best%20Practices%20-%20Coding%20Standards.docx)
* [Traditional Synergy in Visual Studio - CU Wiki](http://echo.cu.net/cuwiki/Traditional_Synergy_in_Visual_Studio)
* [Traditional Synergy in Visual Studio Common Terminology - CU Wiki](http://echo.cu.net/cuwiki/Traditional_Synergy_in_Visual_Studio_Common_Terminology)
* [Debugging (TSVS) - CU Wiki](http://echo.cu.net/cuwiki/Debugging_(TSVS))

# Exercise

1. Using Visual Studio, open the previously created “**Demo**” project.
2. Using Visual Studio, open “**Demo.dbv**”.
3. First we will ensure that only one of the "Salary" or "Rate" fields is available at any point in time, depending on the value selected in the "PayType" field. Edit the program, and do the following:
   * After loading the input window, but before entering the input processing event loop, add the code to disable the "RATE" field in the input window. This reflects the default behavior, as the default value of the "PayType" field is "Annual salary".
   * In the input processing event loop, when you detect a break on the "PayType" field, add code to enable and disable the "SALARY" and "RATE" fields as appropriate. To determine which to do, you will need to look in the variable PayType, which will contain the appropriate data for the current value of the "PayType" field. In addition to enabling and disabling the two fields, clear the field, which is now disabled, and display that data back into the disabled field (as it is not valid for both the "SALARY", and "RATE" fields to have a value at the same time). You will also need to use XCALL I\_NEXT to set the context of the next field processed back to the appropriate field.
   * Once you are happy that the code is correct, simplify the input event loop by placing the new code into an internal subroutine called "pSelectPayType", which you should call from the event loop.
   * Compile, link, and run the application.
4. Next, we will modify the program to have the ability to either create new employee records, or amend existing records. To support this, you will use "[**DemoDataAccess.dbl**](http://jobfunc2.cu.net/Job%20Functions/Programmer/Programmer%20Handbook/Synergy%20Training%20-%20Synergy%20Toolkit/DemoDataAccess.dbl)".
   * Download and save a copy of "[**DemoDataAccess.dbl**](http://jobfunc2.cu.net/Job%20Functions/Programmer/Programmer%20Handbook/Synergy%20Training%20-%20Synergy%20Toolkit/DemoDataAccess.dbl)".
   * Add "[**DemoDataAccess.dbl**](http://jobfunc2.cu.net/Job%20Functions/Programmer/Programmer%20Handbook/Synergy%20Training%20-%20Synergy%20Toolkit/DemoDataAccess.dbl)" to the project.
   * Add the following project references:
     + **ComputersUnlimited.Utilities**
     + **DataAccessLibrary**
     + **SynergyStructures**
   * Add the following import statement to the physical top of “**Demo.dbv**”:
     + **import UIToolkitTraining**
   * "[**DemoDataAccess.dbl**](http://jobfunc2.cu.net/Job%20Functions/Programmer/Programmer%20Handbook/Synergy%20Training%20-%20Synergy%20Toolkit/DemoDataAccess.dbl)" contains a singleton class (that will create the training “dbo.DemoEmployee” table in your default TIMSData database, using your environment, if it doesn’t exist) with the following methods:
     + **DemoDataAccess.Instance.Delete( *id*, *errorFlag* )**
       - ***id*** **(n)**
         * Required, in, Employee.EmployeeId to delete.
       - ***errorFlag*** **(n)**
         * Required, out, Boolean error status:

TRUE – Error occurred.

FALSE – No Error.

* + - **DemoDataAccess.Instance.Lookup( *id*, *errorFlag* )**
      * ***id*** **(n)**
        + Required, in, Employee.EmployeeId to delete.
      * ***errorFlag*** **(n)**
        + Required out, Boolean error status:

TRUE – Error occurred.

FALSE – No Error.

* + - **DemoDataAccess.Instance.Store(** ***dataRecord*,** ***errorFlag*** **)**
      * ***dataRecord*** **(a)**
        + Required, in, Employee record to store.
      * ***errorFlag*** **(n)**
        + Required, out, Boolean error status:

TRUE – Error occurred.

FALSE – No Error.

* + - **DemoDataAccess.Instance.Update(** ***dataRecord*,** ***errorFlag*** **)**
      * ***dataRecord*** **(a)**
        + Required, in, Employee record to update.
      * ***errorFlag*** **(n)**
        + Required, out, Boolean error status:

TRUE – Error occurred.

FALSE – No Error.

* + Create a new internal subroutine called "pLookupEmployee". The role of this routine is to attempt to read an employee record from the database, and display the results into the input window for editing. Attempt to lookup an employee record from the database (using **DemoDataAccess.Instance.Lookup**, and the data in Employee.EmployeeId as the ***id*** value). If the record is found then you should display the contents of the record into the fields of the input window, call the "pSelectPayType" routine to set the "SALARY", and "RATE" fields to appropriate values, disable the "EmployeeID" field, and set the next field to be processed as the "FirstName" field. If the record is NOT found in the database then you can simply disable the "EmployeeID" field before returning to the input event loop. In this routine you will also need to record whether you are creating or amending a record, as you will need this information when you come to update the database after input processing is complete. In addition, you will need to store away the value of Employee.EmployeeId before attempting to read the database, and restore it again if the record was not found.
  + Call "pLookupEmployee" routine when the "EmployeeID" break field is detected.
  + Create a new internal subroutine called "pSaveEmployee". The role of this routine is to update an employee record to the database, or store a new employee record in the database. The routine needs to consider whether you are updating an existing record, or creating a new one, and update the database appropriately. Once the database has been updated, don't forget to re-enable the "EmployeeID" field.
  + Call the "pSaveEmployee" routine when the input set is completed.
  + Compile, link, and run the application.

1. Now we will add code to allow the user to abandon the editing of a record.
   * Create a new internal subroutine called "pAbandonRecord". This routine should consider if an existing record is being updated. If so, it will need to enable the "EmployeeID" field, and initialize the input window, and program data area. If an existing record is NOT being updated, then the routine should cause the input event loop to terminate.
   * Call the new routine when the "CU\_PANIC" menu entry is detected in the input event loop.
   * Compile, link, and run the application.
2. Finally, we will alter the functionality of the "Exit" key (which fires the "CU\_EXIT" menu entry). Currently the exit key simply terminates the input processing event loop - a little harsh if we are in the middle of editing a record!
   * Alter the processing of the "CU\_EXIT" menu entry to use the special "\*DONE\*" function of XCALL I\_NEXT, which will cause input processing to be completed if possible.
   * Compile, link, and run the application.

# Discussion

This is obviously a complex exercise, and the key to success is to take it slowly and step by step. Don’t proceed to another section until you are satisfied that the current section is working.

There is no “correct approach” to input processing – if you find a better way to do something then go with it, so long as the results are similar to those discussed in the above text.