SRP_PreCompiler

The SRP PreCompiler is unlike any other stored procedure in the SRP Utilities collection. It is not a utility you explicitly call. It is, rather, a total enhancement to the BASIC+ language. Indeed, we've come to call the many syntax improvements it provides Enhanced BASIC+.

You do not need the SRP Editor to invoke the SRP PreCompiler. You simply need to include the following line of code at the top of your program:

```
#pragma precomp SRP_PreCompiler
```

This one line instructs the OpenInsight compiler to pass your program to SRP_PreCompiler before passing it on the standard BASIC+ compiler, given the SRP PreCompiler to ability to trans-compile new language features into fully supported ones. These new features are:

- For Each Loops
- Conditional Return Statement
- Service Modules
- Event Modules
- Unit Test Modules
- Unpacking **(Added in 2.1.1)**

Extended BASIC+ Syntax

For Each Loops

For each loops offer a convenient way to walk through a dynamic array from beginning to end. If you have an @FM delimited array, you can do this:

```
For Each Value in MyValues
    Total += Value
Next Value
```

You no longer have to keep track of nesting iterators, extract the current element, and worry about nasty inefficiencies. This represents a massive increase in productivity and readability.

Custom delimiters are supported via the *using* keyword.

```
For Each Value in MyValues using ","
    Total += Value
Next Value
```

You can safely nest For Each loops as well.

```
For Each Field in Array using @FM
    For Each Value in Field using @VM
        Total += Value
    Next Value
Next Field
```

Sometimes, we still need to know the position of the current element. In that case, you can use the *setting* keyword.

```
For Each Value in MyValues using @STM setting Pos
    NewValues<Pos> = Value
Next Value
```

Conditional Return Statement

The SRP PreCompiler extends the Return statement to make it easy to perform a sanity check on the return value. Most people are familiar with the following syntax:

```
If Assigned(Result) else Result = ""
Return Result
```

This code ensures we never accidentally return an unassigned variable. Now, with the help of SRP PreCompiler, we can do this in a single line:

```
Return Result or ""
```

Note that this syntax should only be used for the stored procedure's final Return statement, not for returning from a gosub.

Service Modules

SRP has been a big advocate for organizing code into Service Modules: stored procedures in which the first parameter identifies a service to execute. Normally, this involves branching to gosubs, mapping the input variables, and making sure the stored procedure has enough parameters to cover all the services' needs. Here is a traditional example:

```
Compile function Invoice_Services(Service, Param1, Param2, Param3, Param4, Param5)

Begin Case
    Case Service _EQC "GetCustomerAddress" ; GoSub GetCustomerAddress
    Case Service _EQC "AddSalesTax"        ; GoSub AddSalesTax
    Case 1
        // Service not handled
End Case

If Assigned(Response) else Response = ""
Return Response

GetCustomerAddress:
    AddressType = Param1
    Begin Case
        Case AddressType _EQC "Mailing"  ; Response = CustRec< MAILING_ADDRESS$ >
        Case AddressType _EQC "Shipping" ; Response = CustRec< SHIPPING_ADDRESS$ >
        Case AddressType _EQC "Plant"    ; Response = CustRec< PLANT_ADDRESS$ >
    End Case
return

AddSalesTax:
    Amount  = Param1
    Amount += (Amount * CustRec<TAX_RATE$>)
    Param1  = Amount
return
```

The SRP PreCompiler reduces much of the ceremony involved in this architecture.

First, we save time using parameter placeholders, like so:

```
Compile function Invoice_Services(@SERVICE, @PARAMS)
```

The @SERVICE parameter is required by the SRP PreCompiler to identify it as a service module. The @PARAMS placeholder negates the need to update the parameter list every time you add a new service. The SRP PreCompiler will take care of it for you.

Now, we can simply our branching logic from a long Case statement into a single one:

```
GoToService else
    // Service not handled
end
```

The GoToService keyword uses the @SERVICE parameter to determine what service to jump to. If the service does not exist, it gives you the chance to handle that via the *else* clause. If you don't want to handle it, you can certainly do it in one line.

```
GoToService
```

Now, we write our services. In the past, we did this using standard gosub labels. Now, we use the Service keyword and we list the parameters the service expects:

```
Service GetCustomerAddress(AddressType)
    Begin Case
        Case AddressType _EQC "Mailing"  ; Response = CustRec< MAILING_ADDRESS$ >
        Case AddressType _EQC "Shipping" ; Response = CustRec< SHIPPING_ADDRESS$ >
        Case AddressType _EQC "Plant"    ; Response = CustRec< PLANT_ADDRESS$ >
    End Case
End Service

Service AddSalesTax(Ref Amount)
    Amount += (Amount * CustRec<TAX_RATE$>)
End Service
```

Our services now look like tiny functions within the same stored procedure, complete with their own pass-by-value parameters. When the service is called, we can manipulate the parameters as we see fit without altering the original values passed into the service in the first place. This is good design as most callers expect their parameters to remain unchanged. In the rare case you need to pass back changes to a service parameter, like in the *AddSalesTax* example above, you can precede the parameter with the *Ref* keyword.

You will also notice the Options keyword. This is purely for documentation purposes and will prove

Here is the final code of our service module. It is arguably much cleaner.

```
Compile function Invoice_Services(@SERVICE, @PARAMS)
#pragma precomp SRP_PreCompiler

GoToService

Return Response or ""

Service GetCustomerAddress(AddressType)
    Begin Case
        Case AddressType _EQC "Mailing"  ; Response = CustRec< MAILING_ADDRESS$ >
        Case AddressType _EQC "Shipping" ; Response = CustRec< SHIPPING_ADDRESS$ >
        Case AddressType _EQC "Plant"    ; Response = CustRec< PLANT_ADDRESS$ >
    End Case
End Service

Service AddSalesTax(Ref Amount)
    Amount  = Param1
    Amount += (Amount * CustRec<TAX_RATE$>)
    Param1  = Amount
End Service
```

The SRP PreCompiler also produces meta data about the service when you compile. This metadata can be used by tools such as the SRP Editor to give you coding hints. This makes you much more productive as you can get the information you need about each service as you program.

One such feature, which only the SRP Editor takes advantage of, is options. Options allow you to document a limited set of values for a parameter.

```
Options ADDRESSTYPES = "Mailing", "Shipping", "Plant"
Service GetCustomerAddress(AddressType=ADDRESSTYPE)
    Begin Case
        Case AddressType _EQC "Mailing"  ; Response = CustRec< MAILING_ADDRESS$ >
        Case AddressType _EQC "Shipping" ; Response = CustRec< SHIPPING_ADDRESS$ >
        Case AddressType _EQC "Plant"    ; Response = CustRec< PLANT_ADDRESS$ >
    End Case
End Service
```

The *Options* keyword lets you name a set of options, ADDRESSTYPES in this case, and then assign that name a comma delimited list of quoted or unquoted values. Then, you assign the options, by name, to any parameter that uses those options. In the above example, we assign ADDRESSTYPE options to the AddressType parameter.

Again, this has no effect on how your code functions, but if you are in the SRP Editor, those options will appear in a drop down as you type.

Event Modules

Commuter modules have been around for some time, and they are structured very similarly to services. Typically, gosubs are created for each event, and a large case statement uses the current control and event to branch to the correct handler. Here is a typical example:

```
Compile function MY_WINDOW_EVENTS(CtrlEntId, Event, Param1, Param2, Param3, Param4, Param5)

Window = @Window[1, "F*"]
Control = Field(CtrlEntId, ",", 2, 3)

// Branch to event handler
Begin Case

    Case Control EQ Window
        Begin Case
            Case Event EQ "CREATE"          ; GoSub WINDOW.CREATE
        End Case

    Case Control EQ "MY_EDITFIELD"
        Begin Case
            Case Event EQ "CHAR"            ; GoSub MY_EDITFIELD.CHAR
        End Case

    Case Event EQ "CLICK"
        Begin Case
            Case Control EQ "OK_BUTTON"     ; GoSub PUB_OK.CLICK
            Case Control EQ "CANCEL_BUTTON" ; GoSub PUB_CANCEL.CLICK
        End Case

    Case 1
        // Event not handled

End Case

// Return 1 by default so the event chain continues
If Assigned(EventFlow) else EventFlow = 1
Return EventFlow

WINDOW.CREATE:
    CreateParam = Param1
return

MY_EDITFIELD.CHAR:
    VirtCode = Param1
    ScanCode = Param2
    CtrlKey = Param3
    ShiftKey = Param4
    AltKey = Param5
return

OK_BUTTON.CLICK:
    EventFlow = 0
    End_Dialog(@Window, Data)
return

CANCEL_BUTTON.CLICK:
    Post_Event(@Window, "CLOSE")
return
```

The SRP PreCompiler can clean this up quite nicely. First, we use parameter placeholders in at the top.

```
Compile function MY_WINDOW_EVENTS(CtrlEntId, Event, @PARAMS)
```

This placeholder will allow us to add new event handlers at will without having to remember how many parameters are needed at the top.

Next, we replace the large case statement with a simple branching statement.

```
// Branch to event handler
GoToEvent Event for CtrlEntId else
    // Event not handled
end
```

The GoToEvent statement takes an event name and a control ID, and uses them to branch to the correct event handler. If the handler doesn't exist, then it jumps to the optional *else* clause. Omitting the else close is permitted.

```
// Branch to event handler
GoToEvent Event for CtrlEntId
```

Now, we can rewrite our event gosubs using the Event keyword and supplying named parameters.

```
Event WINDOW.CREATE(CreateParam)
End Event

Event MY_EDITFIELD.CHAR(VirtCode, ScanCode, CtrlKey, ShiftKey, AltKey)
End Event

Event OK_BUTTON.CLICK()
    EventFlow = 0
    End_Dialog(@Window, Data)
End Event

Event CANCEL_BUTTON.CLICK()
    Post_Event(@Window, "CLOSE")
End Event
```

With the SRP PreCompiler, we can rewrite the entire example as so:

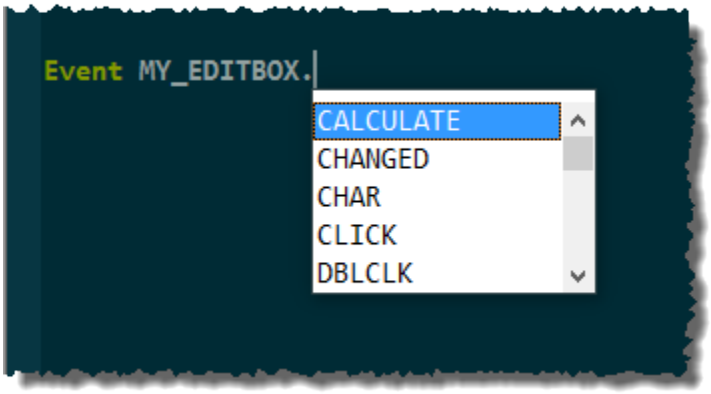
```
Compile function MY_WINDOW_EVENTS(CtrlEntId, Event, Param1, Param2, Param3, Param4, Param5)
#pragma precomp SRP_PreCompiler

// Branch to event handler
GoToEvent Event for CtrlEntId else
    // Event not handled
end

// Return 1 by default so the event chain continues
Return EventFlow or 1

Event WINDOW.CREATE(CreateParam)
End Event

Event MY_EDITFIELD.CHAR(VirtCode, ScanCode, CtrlKey, ShiftKey, AltKey)
End Event

Event OK_BUTTON.CLICK()
    EventFlow = 0
    End_Dialog(@Window, Data)
End Event

Event CANCEL_BUTTON.CLICK()
    Post_Event(@Window, "CLOSE")
End Event
```

Once again, we've simplified the usually tedious task of keeping our commuter module up to date. When a new event handler is need, just add the handler and the SRP PreCompiler will take care of the rest.

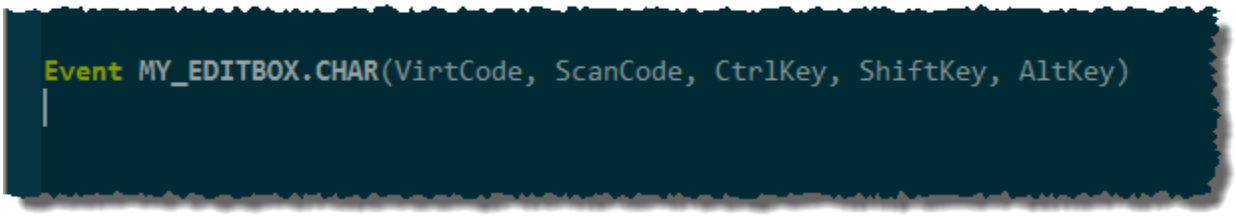
If you use the SRP Editor, you can take advantage of another productivity feature: event autofill. To use it, add the following line at the top of your code:

```
#window MY_WINDOW_NAME
```

Replace MY_WINDOW_NAME with the window you want attached to your commuter module. Once you do this, you will get drop downs when writing event handlers.



Moreover, the SRP Editor will autofill the event's parameters, saving you valuable time.



Unit Test Modules

Unit Test Modules are stored procedures containing multiple related tests.

NOTE: Unit Test Modules are only really useful within the SRP Editor. When a Unit Test Module is compiled, metadata for tests are stored in the system and the tests appear within the SRP Editor UI. This allows you to run any number of unit tests and examine their results.

They have a similar syntax as Service Modules and Event Modules. Here's a sample:

```
Compile function Test_Costumers(@Test)
#pragma precomp SRP_PreCompiler

GotoTest

Return TestResult or 1

Test CalculateAge

        SampleCustomerId = "0001"
    Assert Customer_Services("GetAge", SampleCustomerId) equals 32

End Test
```

The main feature of the Unit Test Module is the Assert statement. It has the following syntax:

```
Assert <actual> equals <expected>
```

This statement is key to making unit tests work. When the evaluation fails, the SRP Editor will display the two values side by side for easy comparison.

Unpacking

IMPORTANT: Unpacking was added in 2.1.1 and is not supported in prior versions.

Unpacking lets you assign elements of a dynamic array into variables *all on a single line*! For example, if you use the SRP_Date Decode service, you'll get an @FM delimited array of information. If you wanted to pull the year, month, and day, you would traditionally need to do this:

```
Info = SRP_Date("Decode", SRP_Date("Today", 1))
Year = Info<1>
Month = Info<2>
Day = Info<3>
```

With unpacking, you can do this:

```
(Year, Month, Day) = SRP_Date("Decode", SRP_Date("Today", 1))
```

The unpacking syntax is activated when you surround your variables in parenthesis. Order matters. The first variable gets item <1>, the second gets item <2>, and so on. You can skip a position using the NULL keyword, an underscore, or just omitting. For example, if you wanted year and day, but not the month, any of these three formats will work:

```
(Year, NULL, Day) = SRP_Date("Decode", SRP_Date("Today", 1))
(Year, _, Day) = SRP_Date("Decode", SRP_Date("Today", 1))
(Year,,Day) = SRP_Date("Decode", SRP_Date("Today", 1))
```

If you need to unpack an array with a delimiter that is not @FM, then you add the USING keyword after the parenthesis:

```
(First, _, Last) using ',' = "John,Seymour,Doe"
```

This syntax is useful for smaller arrays when readability is important to you. This is not useful for parsing large records.

NOTE: You can only unpack into individual variables, not into other arrays or matrices.

Caveats

The SRP PreCompiler will work in any BASIC+ editor, but there is an important caveat to using Enhanced BASIC+ in anything other than the SRP Editor. The SRP Editor recognizes the new syntax and highlights code accordingly whereas the builtin OI Editor will not. Everything will still compile and run, but it will a little harder to read. Moreover, some features of the PreCompiler, such as the metadata, won't impede your code, but you also won't get the most out of your development outside of the SRP Editor.

Another issue with the SRP PreCompiler is that precompiling doesn't produce errors. Thus, if you make an error in the new syntax, the resulting error can be confusing.

Lastly, if you are using SRP PreCompiler, it's best to avoid using variable names that match the keywords *Service, Test,* and *Event.* The SRP Precompiler will do it's best to avoid confusing variables with keywords, but it's not bulletproof. It's better to avoid using them at all if you can help it.