Getting to know AVR for .NET

This document introduces several of the unique and powerful features of ASNA Visual RPG for .NET (AVR). Whether you're a traditional RPG programmer with little or no experience with other languages, or a veteran VB/C# programmer, this document provides an overview of several of AVR's language concepts and syntactical details.

AVR has one foot clearly in the roots of green-screen RPG but the other is firmly planted in .NET. Its version of RPG is sophisticated, highly-evolved version of the language. AVR continues to offer standby RPG idioms and constructs (such as externally described data structures, key lists, parameter lists, record-level operations and program calls). However, it also improves on old-fashioned RPG with a modern, expressive syntax and many new facilities and powers. With AVR, you can build rich Windows applications, powerful browser-based apps, mobile apps, and web services.

This document takes you a quick tour of some of the features and facilities offered by AVR. It's not intended to provide all the details you need for everything it describes; rather, this document's purpose is to introduce you to some of the ways AVR supercharges RPG.

Raising the bar on RPG

AVR's RPG substantially raises the bar on old-time, green-screen RPG. Syntactically, AVR's RPG has been extended to use an indented, block structure. This enhanced syntax lets you write more expressive RPG, removes green-screen RPG's dogged insistence on specific column position, and more naturally lends itself to AVR for .NET's event-driven environment. While AVR doesn't at first look like green-screen RPG, when you learn the basics of its syntax and fully appreciate the power it offers, I think you'll agree that AVR does a good job of merging the best of the old RPG model with the best of what .NET has to offer.

In addition to its numerous syntactical enhancements, AVR brings RPG fully into the world of object-oriented programming. With AVR, you can gracefully implement the three hallmarks of object-oriented programming: encapsulation, polymorphism and specialization. Before we dig into OO detail (that's covered later), lets take a closer look at AVR's syntax and some of its general features.

Green-screen RPG required you to write your code to a very specific columnar scheme (owing back to its reliance on punched cards in the days of its youth). AVR eschews specific columns with a keyword syntax very similar to that used by the iSeries's Command Language (CL). If you're familiar with RPG and CL, it might help to think of AVR as a blend of the best CL's syntax with the power and structure of RPG. (If you're familiar with ILE RPG's full free-format syntax you'll notice a few syntactical similarities between it and AVR as well.)

For example, this snippet of green-screen RPG:

```
C                 Z-ADD  0        TOTBAL
C*
C* Read 10 customer records.
C      1         DO     10
C                 READ   CUST                     50
C      *IN50     ADD    CMBAL    TOTBAL
C                 ENDDO
```

could look like this in AVR:

```
TotalBalance = 0

// Read 10 customer records.
Do FromVal(1) ToVal(10)
    Read Cust
    If (NOT %EOF(Cust))
        TotalBalance = TotalBalance + CMBAL
    EndIf
EndDo
```

This snippet shows several things about AVR's RPG:

- statements are keyworded like CL
- green-screen RPG's column dependence is eliminated
- statements can (and should!) be indented
- whitespace, empty lines, and case don't matter
- two slashes (//) indicate a comments

- RPG built-in functions (BIFS) are supported (you'll later learn that there are also powerful alternatives to BIFs)
- implicit simple variable assignments are allowed (instead of requiring explicit operations like ADD or Z-ADD)
- long identifier names are available
- case doesn't matter—tokens and operations can be expressed in uppercase, lowercase, or any combination thereof.
- operations are always listed before the operation's operands

Another more subtle thing about the AVR snippet above is the phrase 'could look like this.' AVR supports a full complement of backwards-compatible RPG constructs. However, as you'll learn in this document, it also offers powerful alternatives to many long-standing green-screen RPG idioms.

Let's take a closer look at AVR's RPG.

AVR leaves RPG column-dependence behind

AVR's RPG uses CL-like keywords to remove the need for specifically placing code in specific columns. In fact, if you come from an RPG background, you'll notice that AVR was styled almost as though RPG and CL were merged into one language, with CL-like command interfaces provided over all RPG operations. This isn't by accident. AVR retains a great deal of RPG's idioms and personality, but it also needed a more rational, expressive syntax. Given most RPG programmer's experience with CL, blending the best of both worlds was a natural for AVR. AVR applies the 'command interface' concept to virtually all RPG operations as well as RPG specification types such as F-Specs and D-Specs.

AVR uses CL-like rules to govern its use of keywords. Like CL, keywords are optional. For example, if you were to write this code

```
Chain File(Cust) Key(CustKey)
```

or

```
Chain Cust Key(CustKey)
```

or

```
Chain Cust CustKey
```

any of the three lines would compile. AVR's positional rules for keywords would apply and the first value would be assumed to be the from value and the second would be assumed to be the to value. If you omit keywords, the values must be provided in the order in which the RPG operation expects them.

If you specify a given keyword, all keywords following the given keyword must be provided. For example, you could use:

```
Chain Cust Key(CustKey)
```

but you could not use:

```
Chain File(Cust) CustKey
```

The occurrence of the first keyword tells the compiler to ignore positional use from there on. Search AVR's online help for Operation codes to see each operation's keywords.

From a styling perspective, the general rule is to omit the first keyword and include all others for clarity's sake. The additional clarity keywords provide is generally worth the extra typing they require. For example, consider these two lines of code

```
Chain Cust CustNo
```

and

```
Chain Cust Key(CustNo)
```

Semantically, these lines are identical—the compiler accepts either. But the second version is much easier to read. Although the general rule is to omit the first keyword and include all others, there are exceptions. Consider this AVR:

```
Do 1 10 x
```

and

```
Do 1 ToVal(10) Index(x)
```

and

```
Do FromVal(1) ToVal(10) Index(x)
```

These lines show the AVR `Do` loop in action. As you can see, for maximum clarity you will probably want to use all the keywords for a few of the AVR opcodes.

Long-time RPG coders often react negatively to the suggestion that keywords be used to improve code readability (I've heard more than one beginning AVR coder say, "If I wanted to write something as wordy as COBOL, I'd write COBOL!"). You may consider the keyworded version too verbose and to require too much typing. However, remember that it's much harder to read code than it is to write code. Extra care taken to write highly readable code will pay huge dividends over the life of a program.

Veteran green-screen RPG programmers will notice that in most cases AVR's keywords map directly to columns in corresponding green-screen RPG operations. For example, in the Do statement above, the FromVal keyword maps to RPG's factor 1; the ToVal maps to RPG's factor 2; and the Index keyword maps to RPG's result column.

With some RPG operations, such as the `Do` operation, the keywords directly convey the keyword's purpose—without regard for what column the `Do's` operands used to occupy (in other words, the `Do's` keywords `FromVal` and `ToVal` don't reference factor 1 or factor 2 in any way). In others, such as the `Z-ADD` operation, the keywords reflect a more direct relationship with green-screen RPG columns. For example, a keyworded `Z-ADD` to add the value of 8 to variable x could be written as

```
Z-Add F2(8) Result(x)
```

where `F2` is shorthand for the factor 2 column and `Result` is shorthand for the result column. See the AVR help for each operation's exact keywords.

If you are a VB or C# programmer and all this talk about RPG's columns have you confused, don't worry about it. It's your good fortune to be able to learn RPG and not ever have to worry about columns and exactly what goes in each column!

AVR ignores whitespace, empty lines, and case

Not only does AVR remove columnar restrictions, it's also very flexible with regard whitespace, empty lines, and case. For example, you could write

```
MyVar = 9
```

or you could use:

```
MYVAR = 9
```

and AVR would know that both `MyVar` and `MYVAR` are referring to the same variable. Case insensitivity also applies to all other program identifiers (such as routine names, file names and array names). There are two additional important points to make about AVR's case insensitivity:

- Just because case doesn't matter to AVR doesn't mean it doesn't matter to you and other programmers. Strive for diligent consistency with your use of case in your AVR programs.

- Although AVR's language is case insensitive, there will be times when you'll need to pay close attention to case—especially when specifying the keys to collections. This will be covered in more detail later.

AVR is also very liberal about accepting whitespace. For example, you could write

```
Chain Cust Key(CUSTKEY)
```

or (although you probably wouldn't!)

```
Chain     Cust         Key(CUSTKEY)
```

Any additional whitespace is ignored by the AVR compiler. AVR also allows blank lines. Whitespace, or the lack of it, doesn't matter to the compiler; but it matters greatly to programmers trying to read code. The code we write in class will use plenty of whitespace and consistent casing to improve clarity and readability.

Continuing lines

AVR doesn't use an end-of-line mark character as C# or ILE RPG's free-format syntax does with the semi-colon. Each AVR line ends simply with a carriage return. There may be times, for readability purposes, when you want to continue a line to one or more other lines. Consider this `DclDiskFile`:

```
DclDiskFile Cust Type(\*Input) Org(\*Indexed) Prefix(Cust_) File('Examples/CMastNewL2')...
```

you'll often such line written using AVR's line continuation syntax (which, again, borrows from CL):

```
DclDiskFile Cust +
    Type(*Input) +
    Org(*Indexed) +
    Prefix(Cust_) +
    File('Examples/CMastNewL2')
    ...
```

While the second requires more lines, it is much more readable. Continued lines must end with the `+` sign. Continued lines inside expressions are often vexing at first. For example, don't let this line of code trip you up:

```
MyValue = SomethingReallyLong ++
          SomethingElseReallyLong
```

In this case, the first plus sign is part of the expression, the second is the line continuation character.

AVR comments

Anytime AVR encounters two contiguous slashes it assumes they denote a comment area. Used at the beginning of a line, two slashes indicate the entire line is a comment. Used anywhere else in a line indicates that from the slashes to the end of the line is a comment. For example, the snippet below has three comments. One is a full line comment, the other two are partial-line comments.

```
// Read 10 customer records.
Do FromVal(1) ToVal(10)
    Read Cust // Read one record.
    If (NOT Cust.IsEOF) // Record not found.
```

AVR also supports CL/C#-like `/* */` streaming comments. For example, you could also do this:

```
/*
| Read 10 customer records.
*/
Do FromVal(1) ToVal(10)
    Read Cust // Read one record.
    If (NOT Cust.IsEOF) // Record not found.
```

where anything between `/*` and `*/` is considered a comment by the AVR compiler.

AVR statements can (and should!) be indented

Given AVR's whitespace rules, it shouldn't come as a surprise that you can use any indention style you want. You could write this:

```
If (NOT Cust.IsEOF)
    TotalBalance = TotalBalance + CMBAL
EndIf
```

or you could write this:

```
If (NOT Cust.IsEOF)
    TotalBalance=TotalBalance + CMBAL
EndIf
```

However, it is highly recommended that you adopt a rational indention scheme so that the layout of your source code conveys the logic of that code. For example, you could eschew indention and write this:

```
Do FromVal(1) ToVal(RecordsToReadBackwards)
ReadP Cust BOF(BegOfFile)
If (BegOfFile)
Leave
Else
Write CustMem
EndIf
EndDo
```

Or you could do this:

```
Do FromVal(1) ToVal(RecordsToReadBackwards)
    ReadP Cust BOF(BegOfFile)
    If (BegOfFile)
        Leave
    Else
        Write CustMem
    EndIf
EndDo
```

Which stub would you rather try to make sense out of later? The better and more consistent you are with indenting your code, the easier your AVR code will be to read and maintain. Consistent, deliberate use of proper indention is a hallmark of a great AVR program.

AVR data types

AVR has a broad range of data types; some derived from the underlying .NET type system and some from green-screen RPG. A cross-referenced list of AVR's data types is available here. The AVR data types you'll use most frequently are:

- ***Boolean**. The Boolean data type represents a true or false value. Under the covers, `*True` resolves to the numeric value 1 and `*False` resolves to the numeric value zero. AVR provides two keywords, `*True` and `*False`, to use for `*Boolean` testing.

- ***Indicator**. AVR's `*Indicator` data type provides backwards compatibility with green-screen RPG's named indicators. While it is similar to the `*Boolean` data type, `*Indicator` resolves to either character value `'1'` or character value `'0'`. `*Boolean` is the preferred data type for true/false values and should be used over `*Indicator` (this is especially true if you think you'll ever want to share your AVR assemblies with C# or VB programmers).

- *IntegerX*** (where x is 2, 4, or 8). These three integer types represent two-, four-, and eight-byte integer values. RPG programmers should remember that the 2, 4, and 8 refer to data storage bytes, not character positions (as is the case with zoned or packed variables). Thus the `*Integer4` data type has a maximum value of 2,147,483,657 (not 9999 as might be assumed by a green-screen RPG programmer). Generally, you should use the `*IntegerX` data type any time you need a whole number program variable.

- ***String.** Strings are alphameric values of undeclared length. They are similar to green-screen RPG's `*Char` character data type but their length is dynamic and is determined by the variable's content. A string's actual length is determined at runtime. Generally, you should use a `*String` data type any time you need a character program variable. Not only are `*Strings` more convenient than `*Char`s in that strings aren't constrained to a specific length and you don't need to worry about declaring a `*String`'s length.

.NET has nullable versions of several of its data types. With this data type, an integer for example, can either have a numeric value or be null. Nullable types are often handy for working with SQL tables that have nullable columns. While not first-class data types in AVR, with just a little effort you can make .NET's nullable types work in AVR.

AVR's data types do a lot more for you than simply represent a value. As you'll soon see, in AVR for .NET, all variables are instances of objects and these objects bring events, methods, and properties to the party. We'll cover what this means for you in more detail later, but in the meantime, pay close attention as to how data types are used in AVR—you notice that interesting, subtle power lurks for virtually all of AVR's data types. Here's just one example to pique your interest. Consider having `*Packed`, 8, 0 database field named `LoadDate` that stores the date as a numeric value in the format `yyyymmdd`. You could easily format this value like this:

```
LoanDate.ToString('0000-00-00')
```

This code is using the `*Packed` field's `ToString()` method, with a custom numeric formatting string, to render the numeric value as a string with the format given. In this case, zeros mean non-suppressed zero numeric place holders. If `LoanDate`'s value was `20070602` the output from the code above would be `2007-06-02`. Don't worry too much right now about `ToString()` and how its actually formatting the numeric value; rather understand that because virtually all data elements in AVR for .NET are instances of objects, lots of interesting power lurks just below the surface.

Beyond simply adding these (and many other) new data types to your RPG kitbag, you'll soon see that AVR supports all old familiar green-screen RPG data types. And, as alluded to above, you'll see that old friends such as `*Packed`, `*Zoned`, and fixed-length `*Char` character strings aren't just supported, but that AVR highly supercharges them.

Casting and type converions with AVR

.NET's most most basic data type is `System.Object`, which AVR aliases as `*Object`. `*Object` supports all classes in the .NET Framework class hierarchy and provides low-level services to derived classes. It is the ultimate base class of all classes in the .NET Framework; it is the root of the type hierarchy.

One of the purposes of `*Object` is to provide a generic place to store any type of variable. Consider the Windows `ComboBox` for example. It has an `Items` property that is a collection of `*Objects`. This means that any type can be assigned as an drop down item to the `ComboBox`. For example, given a `ComboBox` named `MyComboBox`, you could add a `*String` item like this:

```
MyComboBox.Items.Add('Indiana') // Assign a string to the combo box.
```

To later get the selected item it seems reasonable that you could do this:

```
DclFld SelectedState Type(*String)

SelectedState = MyComboBox.SelectedItem
```

But that fails at compile-time with the message:

```
Cannot convert 'System.Object' to type '*String'
```

When you fetch a value out of a `*Object` you *must* either explicitly convert or cast (a fancy word for convert) it as the type you expect. The first example below converts and the second casts the `SelectedItem`to a string:

```
SelectedState = Convert.ToString(MyComboBox.SelectedItem) // Convert
```

or

```
SelectedState = MyComboBox.SelectedItem *As *String // Cast
```

C# coders are familar with this syntax to cast the SelectedItem to a string:

```
SelectedState = (string)MyComboBox.SelectedItem
```

AVR's `*AS` operator is the functional equivalent to the C# casting show above. VB.NET's `TryCast` and `DirectCast` are VB.NET's special-case ways to cast values.

Let's consider populating the combobox with an instance of an object other than a string.

```
// Because the ComboBox is populated with an object, setting its DisplayMember and
// ValueMember properties to the names of the object's corresponding properties
// tells it what value to display in the ComboBox and what value to fetch with the
// ComboBox's SelectedValue property.
MyComboBox.DisplayMember = 'State'
MyComboBox.ValueMember = 'Abbreviation'

MyComboBox.Items.Add(*New MyComboItem('Indiana', 'IN'))

...

BegClass MyComboItem Access(*Public)
    DclProp State Type(*String) Access(*Public)
    DclProp Abbreviation Type(*String) Access(*Public)
    BegConstructor Access(*Public)
        DclSrParm State Type(*String)
        DclSrParm Abbreviation Type(*String)

        *This.State = State
        *This.Abbreviation = Abbreviation
    EndConstructor
EndClass
```

To fetch the selected item, you might hope this works:

```
DclFld SelectedState Type(MyComboItem)

SelectedState = MyComboBox.SelectedItem
```

It doesn't. It fails at compile-time with the error:

```
Cannot convert 'System.Object' to type `MyComboItem`
```

That this causes an error in .NET has always bugged me. `SelectedState` has clearly been declared as type `MyComboItem` and the compiler knows it because it just told us so in its message; it clearly has all the information it needs to take care of this. No matter, the rule stands:

> When you fetch a value out of a `*Object` you *must* either explicitly convert or cast it as the type you expect.

In this case, .NET doesn't have a built-in method for converting the custom MyComboItem object so it needs to be casted:

```
DclFld SelectedState Type(MyComboItem)

SelectedState = MyComboBox.SelectedItem *As MyComboItem
```

Having done the cast, `SelectedState` is an instance of `MyComboItem` and its public members are availble via this instance.

> Although this article uses the Windows ComboBox for this discussion, web developers also have the need for converting and casting values. The ASP.NET Session is a good example where converting or casting is needed to fetch values stored in the Session object.

Simple variable assignments

Although AVR supports legacy RPG assignment operations such as `ADD`, `Z-ADD` and `MOVE` (along with the myriad related assignment operations), the preferred AVR way to perform variable assignments is with a simple equal statement. For example, you could write this:

```
ZAdd F2(8) Result(x)
```

or

```
ZAdd 8 x
```

However, the preferred way to perform this assignment is to simply use the `=` assignment operator:

```
x = 8
```

AVR's simple assignment operation also has a very interesting implication on the strongly-typed nature of .NET. AVR's equal sign assignment operator is, for the most part at least, the semantic equivalent to green-screen RPG's MOVE operation code (which did double duty as both an assignment operator and, if needed (and possible) a conversion operator. For example, AVR allows this code:

```
DclFld Rate Type(*String)
DclFld NumericValue Type(*Packed) Len(8, 0)

Rate = '45678'
NumericValue = Rate // String assigned to numeric value!
```

In the example above, AVR automatically converts the string value to a numeric value—no explicit conversion is required. (As with green-screen's RPG's MOVE, the code above would fail if the string value contained a non-numeric character.) To RPG programmers familar with green-screen

RPG's MOVE operation, this automatic conversion should seem pretty natural. To VB and C# programmers, though, it's borderline heresy! By its nature, .NET is very type-specific and generally doesn't tolerate the assignment of one data type to another. AVR does indeed follow nearly all the rules for type-specificity that C# and VB (with Option Strict On) do. As you can see, though, AVR's equal assignment operation is more forgiving than what VB or C# allow. For the sake of VB and C# sharp programmers who are especially irked by this violation of type-specifity, relax. You could also write the above code pretty much like you would in VB or C# (In the code below `Convert.ToInt32()` is an intrinsic .NET numeric conversion method that attempts to convert a string to its corresponding integer value):

```
DclFld Rate Type(*String)
DclFld NumericValue Type(*Packed) Len(8, 0)

Rate = '45678'
NumericValue = Convert.ToInt32(Rate)
```

Do note that AVR's implicit assignment conversion only works with the `*String` data type. It does not work with the `*Char` data type. The following code would not compile

```
DclFld Rate Type(*Char)
DclFld NumericValue Type(*Packed) Len(8, 0)

Rate = '45678'
NumericValue = Rate
```

The attempted assignment of a `*Char` data type to a `*Packed` data type would cause the program compile to fail. You can use AVR's implicit assignment conversion with `*String` data types only.

AVR offers alternatives to RPG's traditional indicators

AVR offers several alternatives to both RPG's traditional indicators and its built-in-functions (BIFs). For example, you could write this:

```
Chain Cust NotFnd(*IN50)
If (NOT *IN50)...
```

or this

```
Chain Cust
If (NOT %FOUND(Cust))...
```

or this

```
Chain Cust
If (Cust.IsFound)...
```

Although indicators and BIF's are supported their use for new code is discouraged. The indicator method lends itself easily to problems because the indicator is a global variable (bringing along with it the potential grief that any global variable offers). It's just good defensive coding to avoid indicators wherever you can (and, AVR lets you avoid them everywhere!). In this example, you see that the Cust file has a property called IsFound that indicates the success of the preceding Chain operation. You'll learn later that AVR programs are built of many objects that offer many properties. These object properties are one of the ways you can avoid using indicators in AVR.

Using RPG built-in functions is less problematic from a defensively point of view, but as you read this document, you'll soon discover that AVR and the .NET Framework offer power alternatives to most green-screen RPG BIFs. Having said that, remember that if you want to use BIFs (and even indicators), you certainly can.

For VB/C# programmers, RPG indicators are class-level Boolean variables used to indicate program state. There are 99 of these variables accessible in a special array called *IN (in addition to these 99 indicators, there were also a handful of other special-case indicators). Years ago, before RPG acquired 'structured' operations such as IF and DOW (DoWhile), RPG programmers used global variables to manage program state. As explained in the paragraph above, except for backwards compatibility with old code, indicator usage should generally be considered a deprecated language feature.

AVR allows long variable names

Unlike traditional variable naming rules, AVR allows long variable names. While 15 or 20 characters may be the most rational maximum, AVR allows any length variable name. Long naming applies not only to variable names, but also to control names, arrays and array indexes and subroutine names.

After having lived with the 10-character (and longer ago, 6-character!) restraints green-screen RPG imposes on field names, you'll find AVR's long variable names quite helpful.

AVR opcodes are always listed first

AVR opcodes are always listed before the operation's operands. For example,

```
Chain File(Cust) Key(CustNo)
```

or

```
Do FromVal(1) ToVal(10)
```

This may be initially off-putting to long-time RPG coders, but most quickly realize that having the 'verb' first increases comprehending the code.

AVR programs don't have executable 'mainline' code

In an ILE RPG or RPG/400 program, mainline code is that code provided between the top of the source code and the first subroutine. Any calculation specs provided here were executed once when the program was initially loaded. The essential building block of an AVR program is a class (more about that later) and classes don't have executable mainline code. That may seem quite limiting but you will soon learn that AVR has more than one trick up its sleeve to provide the semantic equivalent of mainline code. (Sneak peak: If you're even conversationally familiar with classes, you may be familiar with what a constructor is. The constructor is one way to achieve 'mainline code' execution.)

Despite not having executable mainline code, AVR does continue the RPG idiom using the 'mainline' area as the place to declare anything that should be globally available to all other routines in that member. Like RPG, you must declare all F-Specs, KeyLists, and constants in this area.

Local variables

Unlike traditional RPG, AVR lets you define program variables scoped only to the subroutine in which they are declared. Locally scoped variables add substantially to the robustness of your code. With locally-scoped variables, you no longer have to worry about other subroutines inadvertently changing a variable's value. With locally scoped variables your subroutines are more loosely coupled to the rest of the routines in your program (and that's a good thing!). When a routine is more loosely coupled to a program, changes made to the routine are less likely to cause problems in other parts of your code.

AVR's `DclFld` operation declares variables. You can use `DclFld` in either the mainline part of the program (AVR has a mainline part of code that, although it can't have any executing code, it is used to declare things global to the rest of the program.) or in subroutines. While AVR still lets you declare variables on the fly (as result fields with operations like `Z-ADD`) that practice is strongly discouraged; it's much better to use DclFld to declare all of your program or routine-level variables.

Briefly a DclFld looks like this:

```
DclFld x Type(*Boolean)
```

The DclFld's lineage owes partly to RPG's D-Spec and partly to CL's `DclVar` operation. The `DclFld` declares a variable's name and its data type. RPG/CL programmers should quickly see its relationship to either RPG's D-Spec and CL's `DclVar`. VB coders would see the above in VB as:

```
Dim x as Boolean
```

And C# coders would see it as

```
boolean x;
```

For VB/C# coders, it's important to remember that AVR does not support block-level variable scoping like VB and C# do. In AVR, variables are scoped either to the class or to a routine.

In addition to local variables, AVR also introduces several new data types. In addition to the expected types of packed, zone, binary, and character (to name a few) AVR also introduces strings, Booleans, and integers (to name a few!). Beyond data types, there's still plenty more to cover with DclFld—as you'll soon see.

Subroutines

Subroutines are the basic build block of an RPG program. In AVR for .NET, you'll see that this is still true—AVR supports subroutines just like green-screen RPG does. For example, this subroutine,

```
BegSr MySubr
    ... do some code
EndSr
```

provides a programming unit just as RPG/400 or ILE RPG programmers are familiar with. It can be called as RPG/400 or ILE RPG would have with

```
ExSr MySubr
```

but AVR also allows this streamlined syntax to call a subroutine:

```
MySubr()
```

The parenthesis denote 'subroutine'—anytime an identifier is followed by parenthesis it is assumed to be a subroutine (or function, which will be covered shortly). Although using ExSr is legal and valid in AVR, this second way is the preferred way to call a subroutine in AVR. `ExSr` should generally be considered a deprecated operation code for new programs.

Why is the second method preferred? It has to with passing arguments to subroutines. AVR lets you pass arguments to subroutines—very much like you may be familiar with passing arguments to an OS/400 program using CALL/PARM. Let that concept soak in for a minute. Empowering subroutines with parameters dramatically reduces the global 'knowledge' a subroutine must have about your program. Using parameters to get data into a subroutine more loosely couples those routines to your program that does their need for using global values to get data. Let's consider an example.

The following subroutine requires two parameters, x and y.

```
BegSr MySubr
    DclSrParm x Type(*Integer4)
    DclSrParm y Type(*Boolean)

    ... do some code
EndSr
```

As you can see, the AVR `DclSrParm` operation is used to define subroutine parameters. The `DclSrParm` operation must follow the `BegSr` operation (or, the soon-to-be explained `BegFunc` operation). The `DclSrParm` operation will covered in more detail later; for now just understand that each DclSrParm describes a parameter's and data type, for a subroutine. You'll also learn later that subroutine parameters can be passed by value or by reference.

`MySubr` can be called two ways. The first of which is the verbose method which uses Exsr and the DclParm operation:

```
ExSr MySubr
DclParm a
DclParm b
```

The `DclParm`'s must be specified in the order in which the target parameters are defined in the subroutine itself (in other words, variable a must be an integer and b must be a Boolean). Like passing parameters to an OS/400 program with Call/Parm, variable names don't have to match between the call and the subroutine definition.

Using the verbose calling method requires a single line for the `Exsr` and additional single lines for each `DclSrParm`. A more succinct way, and the recommended way, of making the call is:

```
MySubr(x, y)
```

There isn't anything wrong with the verbose method. But as you can see, this second method is cleaner and more concise. In either case, a and b are passed as arguments to MySubr. If the subroutine doesn't have parameters, you simply use:

```
MySubr()
```

Take care to provide the trailing, closing parentheses. These parentheses are very important. These parentheses are what indicates to the AVR compiler that this line is calling a routine. Without the parentheses the line wouldn't compile.

C# coders should recognize an AVR subroutine as the semantic equivalent of a C# void function and VB programs should recognize it as a VB procedure.

The previous section said that subroutines are the basic building block of an RPG program. That may have been true in the green-screen RPG world, but in AVR, there is another basic build block called a function. As you'll see in this section, functions are just as important as program building blocks as subroutines are.

Functions

Functions are similar to subroutines, but a function always returns a single value. If you're familiar with RPG BIFs (built-in functions), AVR functions provide a similar capability but you're in control what they do. Let's first look at a built-in function example as a refresher to what a function does. We'll use the `%SUBST` BIF as our example. Consider the code below:

```
If (%SUBST(CustName, 1, 1) = 'A')
    ... Customer name started with 'A'
Else
    ... some other code
EndIf
```

This code uses the RPG %SUBST BIF to determine if the first character of a variable named `CustName` is an `A`. The function returns a string value. Assuming the customer's name did start with an `A`, the `%SUBST` call resolves to an `A`, resulting in the test ultimately looking like this to the compiler:

```
If ('A' = 'A')
```

The concept of 'resolves to' is very important here. The call to `%SUBST` resolved to `A` at runtime. This is an important feature of functions—they are resolved inline and their results can be used in expressions. For example, let's say you wanted to know if a customer's name begins with `AL`. You could use this (silly!) code to do that:

```
If (%SUBST(CustName, 1, 1) + 'L' = 'AL')
    ... Customer name started with 'AL'
Else
    ... some other code
EndIf
```

This illustrates that a function's return value (what it resolves to) can be used in an expression. In other words, the results of a function can be used inline without intermediate variables. In the example above, the test first resolves to this code at runtime

```
If ('A' + 'L' = 'A')
```

Then, immediately to this:

```
If ('AL' = 'AL')
```

In AVR, you'll create functions to return values to your program using your data and the context of your application to determine what should be returned (and why!). Let's take a look at a simple example. The AVR function below calculates a 5% sales tax on a sale amount. The sales tax calculated is the value this function returns.

```
BegFunc CalcSalesTax Type(*Packed) Len(12, 2)
    DclSrParm SalesAmount Type(*Packed) Len(12, 2)

    DclFld TaxAmount Type(*Packed) Len(12, 2)

    TaxAmount = %DECH(SalesAmount * .05, 12, 2)
    LeaveSr TaxAmount
EndFunc
```

Let's take a look at each line of code. The first line declares the function. This does two things: it names the function (`CalcSalesTax`, in this case) and identifies its return type. All functions must have a return type specified in their declaration.

```
BegFunc CalcSalesTax Type(*Packed) Len(12, 3)
```

The next line declares a parameter for the function. In this case, the sales amount is being passed to the function. Functions can have any number of parameters. Note that the parameter types don't have anything to do with the return type. In this case they are the same, but they don't have to be (and often aren't).

```
DclSrParm SalesAmount Type(*Packed) Len(12, 3)
```

The next line declares a return value as in intermediate variable. This line isn't necessary but is added for clarity. A subsequent example will show how this line can be eliminated.

```
DclFld TaxAmount Type(*Packed) Len(12, 3)
```

The next line performs the calculation. In this case the function uses the `%DECH` BIF to calculate the rounded sales tax amount. As with most of AVR's BIFs, the syntax is identical to that of the corresponding ILE RPG function.

```
TaxAmount = %DECH(SalesAmount * .05, 12, 2)
```

The next line returns the value of the function. Every function must have at least one `LeaveSr`. `LeaveSr`'s job is to cause program flow to immediately leave the function, returning the result value specified. AVR's `LeaveSr` opcode is analgous to VB and C#'s `Return` statement.

```
LeaveSr TaxAmount
```

And finally, the function is ended with EndFunc.

```
EndFunc
```

Here's how the preceding function could have been written without the TaxAmount:

```
BegFunc CalcSalesTax Type(*Packed) Len(12, 2)
    DclSrParm SalesAmount Type(*Packed) Len(12, 2)

    LeaveSr %DECH(SalesAmount * .05, 12, 2)
EndFunc
```

In the example above the sales tax is calculated and returned all at once. This style of function saves code and may be appropriate for very simple functions. However for most functions I think you'll find that using an intermediate variable for the return value adds substantially to the clarity of the function.

There is a subtle nuance that the `CalcSalesTax` function illustrates: It receives an input and returns an output. This function knows nothing about the rest of the program, nor does it need to—for all intents and purposes CalcSalesTax is a black box that hides its implementation from the rest of the program. All a consumer need to know about the function is that when you pass it a value, it returns a 5% sales tax. The consumer doesn't know, or care, how the sales tax was calculated.

Long-time RPG coders should ponder the concept of `CalcSalesTax` not having any dependency on global variables. This is a good thing! With `CalcSalesTax`'s action isolated from the rest of the program we know it's 100% safe to make changes to `CalcSalesTax` without causing any unexpected side effects throughout the rest of the program. This black box concept, known also as information hiding or loose coupling is key to quality programs. You'll see this concept revisited many times as you learn about AVR.

Summary

This document introduced you to a number of AVR's unique features. For veteran RPG programmers, concepts were covered that you've never seen in RPG before. There are certainly a lot of new and exciting features in AVR. And, as you'll see as you learn more about AVR, that AVR has taught the old RPG dog many other new dazzling tricks (OO, anyone?). However, you'll also see that AVR, for all its newfound power and capabilities, is surprisingly backwards compatible with many time-honored RPG features and facilities. For example, virtually all of green-screen's RPG file IO model persists. You'll also recognize many other old RPG friends as you read on. For VB/C# coders, this document introduced AVR's syntax and also hinted at the fact that AVR does indeed, semantically if not syntactically, have a lot in common with VB.NET and C#.

As you learn AVR, you'll quickly see that RPG isn't near as old fashioned, or foreign, as you may have once thought.