International Color Consortium®

White Paper #45

Level: Advanced

Date: August 2017

Revised: March 2018

iccMAX MultiProcessingElement Calculator Programming

0 Introduction

This white paper provides an introduction to calculator element script programming as well as using the expanded capabilities of programming with ICC Multi-Process Element Calculator Elements using the XML representation of iccMAX profiles provided by the iccFromXML tool in the RefIccMAX toolset. This white paper assumes familiarity with the iccMAX specification.

1 Calculator element script programming

The Calculator Element uses a stack-based programming model which implicitly accesses a data stack for most operations. Examples of stack-based programming language include: Forth, Factor, Joy, Postscript, and the JavaVirtualMachine.

One feature of stack-based programming models (shared by the iccMAX calculator element) is PostfixNotation, also called ReversePolishNotation. Rather than writing an expression such as

3 + 4

in the calculator element would you encode the operators

3 4 add

A bit unusual, until you get used to it. Postfix notation has the nice property that it doesn't require parentheses for associativity. The following expression is ambiguous:

*3 + 4 * 5*

Is it *(3+4) * 5 = 35*, or *3+(4*5) = 23*? The rules of mathematics and most programming languages say the latter; SmalltalkLanguage says the former. To override these defaults, parentheses must be added. In postfix notation, that's not necessary--you would write either

3 4 add 5 mul

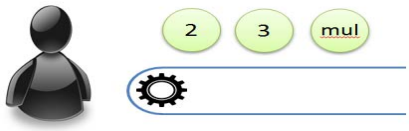
if you mean the first, or

3 4 5 mul add

if you mean the second.

To understand how stack-orientation works, in calculating an expression such as **2** **3** **mul**, consider a simple thought experiment.

Imagine standing at the end of a conveyor belt (the *main function*), onto which have been placed (in sequence) plates (i.e. calculator element operators) marked **2**, **3**, and **mul**. One can take the plate at the end of the conveyor (**2**), but cannot see or take further plates from the conveyor until something is done with the plate just taken. Generally, the way plates can be stored is in a stack, and plates can only be added or removed from atop the stack, not from the middle or bottom. One also has a supply of blank plates (and a marker), and can discard plates (which is permanent). Try to perform the calculation.



Take plate **2** and put it on the stack, then take plate **3** and put it on the stack. Next, take the **mul** plate. This is an instruction to perform. Then, take the top two plates off the stack, multiply their labels (**2** and **3**), and write the result (**6**) on a new plate. Discard the two old plates (**2** and **3**), and put the new plate on the stack. With no more plates remaining on the conveyor, the result of the calculation (**6**) is shown on the plate atop the stack.

This is a very simple calculation. What if a more complex calculation is needed, such as (2 + 3) × 11 + 1? If it is first written in postfix form, that is, **2** **3** **add** **11** **mul** **1** **add**, the calculation can be performed in exactly the same manner and achieve the correct result. The steps of the calculation are shown in the table below. Each column shows an input element (the plate at the end of the conveyor), and the contents of the stack after processing that input.

Operators	2	3	add	11	mul	1	add
Stack	2	3 2	5	11 5	55	1 55	56

After processing all the input, the stack contains **56**, which is the answer.

From this, the following can be concluded: a strictly stack-based programming language has only one way to handle data, by taking one piece of data from atop the stack, termed *pop*ping, and putting data back atop the stack, termed *push*ing.

However, the calculator element is not strictly stack-based. It is also not a general purpose programming language, but rather it is a low-level scripting language that enables one to encode color transforms with close affinity to processing elements within a multiProcessingElement tag type.

The main function of the calculator element contains a list of operations to perform with each operation generally able to do one or more of the following:

- take data off the stack
- directly perform some computational operation
- apply a processing sub-element
- get data from a CMM environment variable
- place data onto the stack
- get data from input channels
- store data to output channels
- store data to indexed memory
- retrieve data from indexed memory
- manipulate stack values
- conditionally select operations to perform

Additionally, each calculator element operation has a 32-bit data parameter that is used to provide control and selection behavior of the operation. This allows most operations to be performed on vectors or groups of data values using only a single operator. This also provides greater security and predictable behavior since selection control is part of the operator (rather than being part of data on the stack).

Calculator elements are encoded as binary structures as described in Clause 11.2.1 of the iccMAX specification where the list of all available operators as well as descriptions of operations relative to stack values is provided. To make calculator element programming easier the Main Function can have a textual representation as described in Annex F of the iccMAX specification. Additionally, the XML representation of iccMAX profiles (provided by XML libraries in RefIccMAX) incorporates and extends this textual representation to provide a more robust higher level method of dealing with calculator elements.

Note: A worked example of using a calculator element script to transform Adobe RGB to XML can be found in Annex A of this document.

2 Basic Representation of Calculator elements as XML

The iccFromXML tool provides the ability to directly encode a Multi-ProcessElement (MPE) calculator element defined in Clause 11.2.1 of the iccMAX specification as XML text, with the operations defined using the textual representation defined in Appendix F of the iccMAX specification. The general XML encoding of a calculatorElement used in a MultiProcessElement is as follows:

```
<CalculatorElement InputChannels="in" OutputChannels="out">

    <SubElements>
        <XmlElemName0 InputChannels="in0" OutputChannels="out0">
            <-- SubElement 0 definition goes here -->
        </XmlElemName0>
        <XmlElemName1 InputChannels="in1" OutputChannels="out1">
            <-- SubElement 1 definition goes here -->
        </XmlElemName0>
        …
    </SubElements>

    <MainFunction>Textual Representation of Operations</MainFunction>

</CalculatorElement>
```

With the following considerations:

- Like all XML representations of MPEs, the calculator element specifies both the number of channels coming into the processing element (*in*) as well as the number of channels coming out of the processing element (*out*). These channels are accessed in the MainFunction using the **in** and **out** calculator operators with optional **[***position***]** or **[***position,size***]** designations to access/set specific channels.

- The SubElements section is optional and required if sub-element processing is utilized by the calculator script defined in the <MainFunction> block. The CalcElement operators and associated XML element encoding names (*XmlElemNameX*) used to define sub-sections of the <SubElements> block are outlined in Table 1.

Table 1 - Element operators and XML Encoding names

Operator	XML Encoding Name
calc	CalculatorElement
clut	CLutElement or ExtCLutElement
curv	CurveSetElement
fJab	JabToXYZElement
mtx	MatrixElement
tint	TintArrayElement
tJab	XYZToJabElement

Use of each of these operators in the <MainFunction> block are followed by a [*position*] designation (I.E. **mtx**[0] corresponding to a <MatrixElement> as the first sub-element) which identifies the zero indexed element to invoke that is defined in sequential order in the <SubElements> block. Alternatively the **elem**[*position*] operator can also be used to invoke the sub-element at the designated position in the associated sub-element array. Additional elements that can be invoked using the **elem** operator include the EmissionCLutElement, EmissionMatrixElement, EmissionObserverElement, InvEmissionMatrixElement, ReflectanceCLutElement, and ReflectanceObserverElement.

- The textual representation of the operators represents a sequence of operations. Operations are separated by whitespace. A sequence of operations begin with the pseudo-operator **{** and end with the pseudo-operator **}**. Conditional operations (**if**, **else**, **sel**, **case** and **dflt**) also use these pseudo-operators to designate streams of operations.

- Temporary memory is accessed using the **tget**, **tput**, and **tsav** operators with optional **[***position***]** or **[***position,size***]** designations to accesses/set specific memory elements. The meaning of temporary memory element usage is entirely up to the programmer. For performance purposes, temp memory elements should be initialized before reading from them.

The XML structure for <CalculatorElement> has direct one to one relationships with the binary structures stored in a MPE calculator element in an iccMAX based profile. With this structure in place one can directly program MPE calculator elements in an iccMAX profile. However, due to the low level nature of the calculator element there are a few challenges in programming with a calculator element.

1. Temporary memory variables are indexed by position. There is no method of associating a name (or mnemonic) to temporary variable positions. Confusion between indices can easily occur as a result, and the meaning of what variables are for or how they are used cannot be directly be deduced from casual looking at the MainFunction script. This can result in a much more challenging process to get bug-free calculator script code.

2. Likewise, sub-elements are also indexed by position. There is no method of associating a name (or mnemonic) to sub-elements. Confusion between sub-element indices can occur as a result and the meaning of what sub-elements are for or how they are used cannot be directly deduced from looking at the MainFunction script. This can result in a much more challenging process to get bug-free calculator script code.

3. Every operation to be performed needs to be explicitly placed in the MainFunction. Sequences representing common functional operations need to be explicitly duplicated in the function. There is no way to associate a simple sequence as a named operation (macro) to be performed. Having such sequences in the MainFunction makes it larger and more difficult to understand higher level concepts. Using a **calc** sub-element can help with this, but there are performance overheads.

4. The MainFunction is monolithic, and therefore unwieldy. Higher level programming languages allow for functional libraries to be developed providing for code reuse and simpler development processes.

From an execution standpoint the MPE Calculator provides a low level mechanism for defining and executing color transforms. From a conceptual point of view it represents a low level transform assembly language with direct access to memory locations on functional operations. However, for easier development, more flexibility and readability a higher order macro assembler approach can be leveraged to help overcome the above-mentioned challenges.

3 Extended Representation of Calculator elements as XML

3.1 Overview

Fortunately the process of parsing XML to create binary iccMAX profiles allows for multiple passes to be incorporated to overcome the challenges mentioned in the previous section. The IccLibXML library utilized by the iccFromXML tool in the RefIccMAX project provides for extend support of defining calculator elements over the basic support to directly creating calculator elements on a one to one relationship with the binary structures defined by the iccMAX specification. These extensions make it easier to develop calculator elements with the flexibility to utilize code libraries that are more readily understandable. Thus from a conceptual point of view iccFromXML provides a macro assembler for calculator element development. This is done by allowing for variables and sub-elements to be named, named macros to be defined that encapsulate common sequences of operations, and the ability to define groupings of variables, sub-elements, and macros as transform libraries that can be developed, imported, and re-used. In the end though, these extensions are distilled by iccFromXML down to the same basic representation required by the iccMAX specification thus providing a level of obfuscation to the calculator element code used to generate the binary profile.

Note: A worked example of using extended XML with a calculator element to perform ink layer overprint simulation can be found in Annex B of this document.

3.2 Extended structure

The general XML encoding of a CalculatorElement used in a MultiProcesElement is as follows:

```
<CalculatorElement InputChannels="in" OutputChannels="out" InputNames="…" OutputNames="…">
        <Imports> … </Imports>
        <Variables> … </Variables>
        <Macros> … </Macros>
        <SubElements>… </SubElements>

        <MainFunction>Extended Textual Representation of Operations</MainFunction>
</CalculatorElement>
```

The InputNames and OutputNames attributes are optional and used to provide names for input and output channels. This is described in the next section.

The <Imports>, <Variables>, <Macros>, and extended <SubElements> XML blocks are optional and used as needed in defining the Main Function. The description and use of each of these blocks are described in the following sections.

The extended MainFunction is parsed in multiple passes by first flattening macro usage into sequences of operations while performing sub-element redirection and variable assignment and redirection. After this is performed, the resulting low level operations list is converted to binary sub-elements and function operations of a binary Calculator Element within an iccMAX profile.

An important part of this process is the definition of what actually is included in the final binary. Variables, macros and named sub-elements that are not referenced by the MainFunction (or any macros directly or indirectly that it calls) are not included in the final binary. This allows for large libraries of varied functionality to be defined and only those parts that are essential to the functionality of the CalculatorElement will be included in the profile.

3.3 Named Input and Output Channels

Normally input and output channels are accessed using the **in** and **out** operators specifying the channel index position and number of channels to copy. For example: **in**[3] places the input channel value at index 3 onto the stack, and out[6,3] take three elements from the stack and places them output channels starting at position 6.

With the extended representation of XML calculator elements, optional attributes can be added to the <CalculatorElement> block to define names for input channels and/or output channels. Input channel names are specified with an InputNames="…" attribute, and output channel names are specified with an OutputNames="…" attribute. The contents of the text associated each attribute defines space delimited channel names in similar fashion to the way that member variables are defined for a structure variable (see Section 2.5.1). Additionally, similar to the declaration of vector members of a structure variable, vector named channels can be declared by immediately putting the vector size in square brackets immediately after the local vector name.

The use of named channels is performed by placing a channel specifier within curly brackets immediately after the **in** or **out** operators. Channels defined in InputNames shall only be associated with the **in** operator, and channels defined in OutputNames shall only be associated with the **out** operator. Therefore, position and vector sizes of channels in InputNames can be different to the position and vector sizes of identically named channels in OutputNames. Names in InputNames

and OutputNames cannot be used by **tput**, **tget**, and **tsav** operators to access input or output channels.

An example of using named input channels and output channels is as follows:

```
<CalculatorElement InputChannels="7"
                   OutputChannels="6"
                   InputNames="C M Y K Lab[3]"
                   OutputNames="a b C MY[2] L">
    <MainFunction>
{
  in{C}          ;place input channel 0 onto stack
  out{C}         ;place stack value into output channel 2
  in{Lab[1],2}   ;place 2 input channels starting at index 5 (i.e. 4+1) onto stack
  out{a,2}       ;place 2 values on stack into output channels starting at index 0
  in{M}          ;place input channel 1 onto stack
  in{Y}          ;place input channel 2 onto stack
  out{MY}        ;place 2 values on stack into output channels starting at index 3
  in{Lab[0]}     ;place input channel 4 onto stack
  out{L}         ;place value on stack into output channel 5
}
    </MainFunction>
</CalculatorElement>
```

3.4 Calculator element imports

The <Imports> block is the first thing parsed when parsing a Calculator element. This allows for one or more IccCalcImport files to be imported to define Variables, Macros, and SubElements that can be utilized either within the Macros and MainFunction of the CalculatorElement or by Macros defined within an import file.

3.4.1 Import encoding

The general encoding of an Import block is as follows:

```
<Imports>
    <Import Filename="file_specifier_1.xml"/>
    <Import Filename="file_specifier_2.xml"/>
    …
</Imports>
```

Each of the import *file_specifier_X.xml* entries defines a path to an XML file containing an IccCalcImport XML structure. When parsing the <Imports> section, each of the files specified by the contained <import> blocks is opened and parsed sequentially. The general encoding of one of these files is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<IccCalcImport>
    <Imports>…</Imports>
    <SubElements>…</SubElements>
    <Variables>…</Variables>
    <Macros>…</Macros>
</IccCalcImport>
```

The encoding of an IccCalcImport mirrors that of a CalculatorElement, and the sub-blocks within an IccCalcImport file have the same meaning and encoding as similarly named sub-blocks within an extended CalculatorElement with a few exceptions. These being that all SubElements in an IccCalcImport file shall be named, and Variables shall not have offsets defined.

3.4.2 Import usage

The Imports block of an IccCalcImport file is parsed before any of blocks in an IccCalcImport file. Recursion of import files shall be short circuited, and recursion of imports involved with sub calculator elements shall not be allowed.

When parsing an import file, the named SubElements, Variables, and Macros within an IccCalcImport file are added to the lists of named SubElements, Variables and Macros of the containing CalculatorElement. Thus they have global scope within the context of the Calculator Element's MainFunction and any macros that are directly or indirectly called.

Note: The SubElements, Variables, and Macros of a CalculatorElement are not accessible by a calculator based sub-element.

3.5 Calculator element variables

The use of named variables within a calculator element provides for both greater readability, as well as simplification of the calculator script operations. Variables can be defined as named regions of Calculator Temporary Memory with flexibility in specifying size, position, and structure. Though variables' positions can be explicitly assigned within temporary memory, it is generally not recommended.

3.5.1 Variable encoding

CalculatorElement variables are defined in <declare> sub-blocks of a <Variables> block. Variable names can be made up of any sequence of alpha-numeric characters (A-Z, a-Z, 0-9, and underscore '_'). Variable names shall not begin with numeric characters. Names of all variables defined within a Calculator or IccCalcImport file are global to the Calculator Element.

There is flexibility in how variables can be declared as shown in the following example:

```
<Variables>
        <Declare Name="myVar"/>
        <Declare Name="myVector" Size=6/>
        <Declare Name="myStruct">m1 m2[3] m3</Declare>
</Variables>
```

In the first declare block a single data element variable is defined (myVar).

In the second declare block a vector variable containing 6 data elements is defined (myVector).

In the third declare block a structure variable containing 5 data elements (myStruct) with three members. The first member (m1) has one data element at offset 0 of myStruct, the second member (m2) has 3 data elements at offset 1 of myStruct, and the third element has 1 data element at offset 4 of myStruct. Member variables of a structure are defined as space delimited values in the content of a declare block. Member vectors can be defined by placing the vector size in square brackets immediately after the member name. Member names must be unique to their containing structure, and have the same encoding restrictions as variable names.

Adding an attribute specifying Position="*x*" in any of the above variable declare blocks indicates the address (*x*) for the variable in calculator temporary memory storage. When a position is not explicitly provided then one will be determined automatically as the MainFunction and macros that use such

variables are parsed. Assignment of addresses for variables to calculator temporary memory storage is only performed for variables that are actually used by either the MainFunction or macros that are directly or indirectly called by the MainFunction. Additionally, variables with unassigned position specification are likely to be positioned after the last positioned variable that is used.

Variables inside of IccCalcImport files shall not have position specifiers as this allows them to be easily repurposed.

3.5.2 Variable usage

Named variables are incorporated into a calculator script code as an extension to parsing of the **tget**, **tput** and **tsav** operators. Normally these operators specify position and size within the temporary memory block. For example: **tget**[5] puts the data value located at position 5 in calculator temporary storage, and **tput**[6,3] take three elements from the stack and places them into calculator temporary memory storage starting at position 6.

The use of named variables is performed by placing a variable specifier within curly brackets immediately after one of these three operators. Thus in relation to the example variables declared in the previous section:

tget{*myVar*} places the single value associated with *myVar* onto the stack.

tput{*myVector*} takes 6 values off the stack and places them into the calculator temporary memory associated with *myVector*.

tsav{*myStruct*} copies the top 5 values on the stack into the calculator temporary memory associated with *myStruct*.

Parts of either a vector or structure can also be selectively referenced as shown in the following additional examples:

tget{*myVector*[3]} puts a single value from position 1 of *myVector* onto the stack.

tput{*myVector*[4,2]} takes two values off the stack and places them into *myVector* starting at position 4.

tsav{*myStruct.m3*} copies the top single value on the stack into the *m3* member of *myStruct*.

tsav{*myStruct.m2*} copies the top three values on the stack into the *m2* member of *myStruct*.

tget{*myStruct.m2*[2]} places a single value from position 2 of the *m2* member of *myStruct* onto the stack.

tput{*myStruct.m2*[1,2]} takes two values off the stack and places them starting at position 1 of the *m2* member of *myStruct*.

3.6 Calculator element macros

The <Macros> block allows for macros to be defined that provide sequences of operations to be injected or inserted into the operation sequence of a calling sequence. They are invoked with the **call** pseudo-operator. As they represent an insertion of operations there is no extra overhead

associated with using them, other than adding additional operators to the operation sequence. Macros are allowed to make calls to sub-macros although macro recursion is not allowed. Macros are also allowed to define and use local variables that only have scope within the macro.

3.6.1 Macro encoding

The general encoding of a Macros block is as follows:

```
<Macros>

        <Macro Name="macro1">Text defining macro1 operator sequence</Macro>
        <Macro Name="macro2">Text defining macro2 operator sequence</Macro>      …
</Macros>
```

The content text for each macro block contains a sequence of operations to inject when the macro is invoked. The content of operators for a macro is application specific. Named constants can be defined by declaring a macro that just places a constant value on the stack. For example:

```
        <Macro Name="pi">3.141592653590</Macro>
```

3.6.2 Macro usage

Macros are invoked with the **call** pseudo-operator, and they represent an injection of the macro's operations into the operation sequence of a calling sequence. To better understand this, consider the following CalculatorElement definition that functionally adds the value of 42 to the input using macros:

```
<CalculatorElement InputChannels="1" OutputChannels="1">
        <Macros>
                <Macro Name="odd">1 3 5 5 3 1</Macro>
                <Macro Name="evenoddeven">2 4 6 call{odd} 6 4 2</Macro>

        </Macros>

        <MainFunction>{ in[0] call{evenoddeven} sum(13) out[0] }</MainFunction>
</CalculatorElement>
```

When this MainFunction is parsed and encoded into a binary CalculatorElement in an iccMAX profile the function is flattened by replacing macro calls with the operators associated with the macro. In this case the **call**{*evenoddeven*} is replaced by the operators defined for macro *evenoddeven*. However, since the *evenoddeven* macro also contains a macro invocation using **call**{*odd*} the operators for the odd macro are inserted at the appropriate place in the operators for *evenoddeven*.

Thus the macro flattening process for the above CalculatorElement is equivalent to defining a CalculatorElement with no macros as follows:

```
<CalculatorElement InputChannels="1" OutputChannels="1">
        <MainFunction>{ in[0] 2 4 6 1 3 5 5 3 1 6 4 2 sum(13) out[0] }</MainFunction>
</CalculatorElement>
```

The resulting binary MPE calculator element encoding in an iccMAX profile for both of these Calculator elements is identical.

3.6.3 Macro local variables

When defining macros it is often desirable to be able to use variables to store temporary intermediate results. Though variables declared using a <Variable> block can be used within a macro, such variables are not temporary but global in scope of the CalculatorElement. When defining a macro one can define variables that are local to the macro with scope only defined within the context of the macro. Macro variables are also associated with calculator temporary memory storage. However their allocation is defined separately from <Variable> block variables and are reused by macros as they are sequentially used. The contents of macro local variables is therefore not well defined and they should be initialized before using them.

Macro local variables are declared by adding a Local="…" attribute to the <Macro> block specification. The contents of the text associated with the local attribute defines space delimited local variable names in similar fashion to the way that member variables are defined for a structure variable (see Section 2.5.1). Additionally, similar to the declaration of vector members of a structure variable, a local vector can be declared by immediately putting the vector size in square brackets immediately after the local vector name.

The use of local variables when defining an operation sequence is indicated by placing an @ before the local variable name using the **tget**, **tput**, and **tsav** operators. Indexing sub ranges of local variables is identical to indexing of variables. Additionally, because local variables are identified using the @ symbol, global variables can have the same name as local variables without a naming conflict.

Consider the following macro example which clamps 3 values on the stack between a range of two values placed on stack aftwerwards (invoked by placing 5 values total on stack before using **call**{*clamp3*}):

<Macro Name="clamp3" Local="range[2]">tput{@range} tget{@range[1]} copy[1,2] vmin(3) tget{@range[0]} copy[1,2] vmax(3)</Macro>

In this case the **tput** operator takes two values off the stack and places them into the local vector named *range*. Then the second range value is placed on the stack, duplicated two times, and a vectorised ***min*** function (of size 3) is applied. Then the first range value is placed on the stack, duplicated two times, and a vectorised ***max*** function (of size 3) is applied. This results in the 3 values remaining on the stack to be clamped to the desired range.

Alternatively this could be done using two separate local variables as follows:

<Macro Name="clamp3" Local="lower upper">tput{@upper} tput{@lower} tget{@upper} copy[1,2] vmin(3) tget{@lower} copy[1,2] vmax(3)</Macro>

3.7 Calculator element named sub-elements

The ability to invoke embedded MPE processing elements within a calculator element script leverages the power of optimized transforms within the control of the script. The use of named sub-elements within a calculator element also provides for both greater readability, as well as simplification of the calculator script operations. Therefore, the use of named sub-elements is generally recommended.

3.7.1 Named sub-element encoding

Any sub-element defined within a <SubElement> block can be given a name by adding a Name="…" attribute to the sub-element's definition block. Sub-Element names can be made up of any sequence of alpha-numeric characters (A-Z, a-Z, 0-9, and underscore '_'), and shall not begin with numeric characters. Names of all sub-elements defined within a Calculator or IccCalcImport file are global to the Calculator Element. Naming of sub-elements is demonstrated in the following example:

```
<SubElements>
        <CurveSetElement Name="applyGamma" InputChannels="3" OutputChannels="3">…</CurveSetElement>
        <MatrixElement Name="RGBtoXYZ" InputChannels="3" OutputChannels="3">…</MatrixElement>

</SubElements>
```

As mentioned in Section 1, unnamed sub-elements are placed in the calculator element's sub-element array in the order that they are found in the <SubElements> block. All unnamed sub-elements are placed in the sub-element array regardless of whether they are accessed by the MainFunction Script

Named sub-elements (unlike unnamed sub-elements) do not have a fixed position in the sub-element array. They are assigned a position in the sub-element array directly after any unnamed sub-elements as they are accessed by the MainFunction script (or any macro that is directly or indirectly called from the Main Function). This allows for declarations of named sub-elements to be re-ordered in a <SubElement> block without having to make changes to the calculator elements script code. If a named sub-element is not accessed directly or indirectly by the MainFunction then the sub-element will not be included in the binary iccMAX calculator element sub-element array, and is therefore not assigned a position.

All sub-elements defined inside of the <SubElements> block of IccCalcImport files shall have names associated with them. This allows for these sub-elements to be repurposed in the form libraries of sub-elements.

3.7.2 Named sub-element usage

Named sub-elements are incorporated into calculator script code as an extension to parsing of the sub-element invocation operators like **elem** and the operators identified in Table 1. Normally these sub-element invocation operators specify the zero indexed position of the sub-element to invoke in the sub-element array. For example: **elem**[5] invokes the sixth sub-element in the sub-element array.

The use of named sub-elements is performed by placing the name of the sub-element to be invoked within curly brackets immediately after the sub-element invocation operator. An example of invoking the elements defined in the previous section is demonstrated in the following MainFunction defintion:

```
<MainFunction>{ in[3] curv{applyGamma} mtx{RGBtoXYZ} out[3]}</MainFunction>
```

Annex A - Example calculator element script

A.1 Introduction

To get started with calculator programming consider defining a transform that goes from AdobeRGB to tristimulus XYZ values using a single calculator element within a multiProcessElementsType based tag.

> *Note: The test profile argbCalc.icc profile in the RefIccMAX testing suite (defined by the argbCalc.xml file) provides one such implementation.*

From the AdobeRGB specification there are two basic steps in the process of converting RGB to XYZ. The first step is the application of a gamma power function to the RGB values to convert to linear RGB.

$R = R'^{\,2.19921875}$, $G = G'^{\,2.19921875}$, $B = B'^{\,2.19921875}$

The second step applies a matrix transform that converts linear RGB to XYZ values.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.57667 & 0.18556 & 0.18823 \\ 0.29734 & 0.62736 & 0.07529 \\ 0.02703 & 0.07069 & 0.99134 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

This can be represented as a MainFunction of a Calculator Element as follows:

```
{
in(0,3) 2.19921875 gama(3)
tsav(0,3)
0.5766686480064009 0.1855585057676510 0.1882277564620142 mul(3) sum(3)
tget(0,3)
0.2973431495063504 0.6273623341556915 0.0752878092554091 mul(3) sum(3)
tget(0,3)
0.0270330176683369 0.0706883571748087 0.9913426165633547 mul(3) sum(3)
out(0,3)
}
```

A.2 Demonstration

The following examples describe how the above script operations modify the contents of the stack, memory storage, and output channels for two sets of RGB input values.

Input
1.0, 1.0, 1.0

Operation	Description	Stack	Memory	Output
	Initial State			
in(0,3)	Place first three input channels on stack	1.0, 1.0, 1.0		
2.21992	Push 1 data value on stack	1.0, 1.0, 1.0, 2.21992		
gama(3)	Apply gamma value to three values on stack	1.0, 1.0, 1.0		
tsav(0,3)	Copy top three elements of stack into memory at index 0	1.0, 1.0, 1.0	1.0, 1.0, 1.0	
0.5767 0.1856 0.1882	Place 3 data values on stack	1.0, 1.0, 1.0, 0.5767, 0.1856, 0.1882	1.0, 1.0, 1.0	
mul(3)	Multiply top three values by next top three values on stack and place results on stack	0.5767, 0.1856, 0.1882	1.0, 1.0, 1.0	
sum(3)	Add top three values and place result on stack	0.9505	1.0, 1.0, 1.0	
tget(0,3)	Place three memory values starting at index 0 onto stack	0.9505, 1.0, 1.0, 1.0	1.0, 1.0, 1.0	
0.2973 0.6274 0.0753	Place 3 data values on stack	0.9505, 1.0, 1.0, 1.0, 0.2973, 0.6274, 0.0753	1.0, 1.0, 1.0	
mul(3)	Multiply top three values by next top three values on stack and place results on stack	0.9505, 0.2973, 0.6274, 0.0753	1.0, 1.0, 1.0	
sum(3)	Add top three values and place result on stack	0.9505, 1.0	1.0, 1.0, 1.0	
tget(0,3)	Place three memory values starting at index 0 onto stack	0.9505, 1.0, 1.0, 1.0, 1.0	1.0, 1.0, 1.0	
0.0270 0.0707 0.9913	Place 3 data values on stack	0.9505, 1.0, 1.0, 1.0, 1.0, 0.0270, 0.0707, 0.9913	1.0, 1.0, 1.0	
mul(3)	Multiply top three values by next top three values on stack and place results on stack	0.9505, 1.0, 0.0270, 0.0707, 0.9913	1.0, 1.0, 1.0	
sum(3)	Add top three values and place result on stack	0.9505, 1.0, 1.0891	1.0, 1.0, 1.0	
out(0,3)	Store 3 values on stack to output channels starting at index 0		1.0, 1.0, 1.0	0.9505, 1.0, 1.0891

Input
0.5, 0.5, 0.5

Operation	Description	Stack	Memory	Output
	Initial State			
in(0,3)	Place first three input channels on stack	0.5, 0.5, 0.5		
2.21992	Push 1 data value on stack	0.5, 0.5, 0.5, 2.21992		
gama(3)	Apply gamma value to three values on stack	0.2196, 0.2196, 0.2196		
tsav(0,3)	Copy top three elements of stack into memory at index 0	0.2196, 0.2196, 0.2196	0.2196, 0.2196, 0.2196	
0.5767 **0.1856** **0.1882**	Place 3 data values on stack	0.2196, 0.2196, 0.2196, 0.2196, 0.5767, 0.1856, 0.1882	0.2196, 0.2196, 0.2196	
mul(3)	Multiply top three values by next top three values on stack and place results on stack	0.5767, 0.1856, 0.1882	0.2196, 0.2196, 0.2196	
sum(3)	Add top three values and place result on stack	0.2088	0.2196, 0.2196, 0.2196	
tget(0,3)	Place three memory values starting at index 0 onto stack	0.2088, 0.2196, 0.2196, 0.2196	0.2196, 0.2196, 0.2196	
0.2973 **0.6274** **0.0753**	Place 3 data values on stack	0.2088, 0.2196, 0.2196, 0.2196, 0.2973, 0.6274, 0.0753	0.2196, 0.2196, 0.2196	
mul(3)	Multiply top three values by next top three values on stack and place results on stack	0.2088, 0.0653, 0.1378, 0.0165	0.2196, 0.2196, 0.2196	
sum(3)	Add top three values and place result on stack	0.2088, 0.2196	0.2196, 0.2196, 0.2196	
tget(0,3)	Place three memory values starting at index 0 onto stack	0.2088, 0.2196, 0.2196, 0.2196, 0.2196	0.2196, 0.2196, 0.2196	
0.0270 **0.0707** **0.9913**	Place 3 data values on stack	0.2088, 0.2196, 0.2196, 0.2196, 0.2196, 0.0270, 0.0707, 0.9913	0.2196, 0.2196, 0.2196	
mul(3)	Multiply top three values by next top three values on stack and place results on stack	0.2088, 0.2196, 0.0059, 0.0155, 0.2177	0.2196, 0.2196, 0.2196	
sum(3)	Add top three values and place result on stack	0.2088, 0.2196, 0.2392	0.2196, 0.2196, 0.2196	
out(0,3)	Store 3 values on stack to output channels starting at index 0		0.2196, 0.2196, 0.2196	0.2088, 0.2196, 0.2392

Annex B - Worked XML example

B.1 Introduction

Working examples of using extended calculator programming can be found in the Testing files of the RefIccMAX repository. The following example is provided for informational purposes.

B.2 Seven channel ink layering model

The following example uses named sub-elements to demonstrate encoding an ink laydown model for a transform that converts CMYKOGP ink values to XYZ values as an extended calculator element. The input channel indexing order is Cyan=0, Magenta=1, Yellow=2, Black=3, Orange=4, Green=5, Purple=6. The ink laydown order is Yellow, Orange, Magenta, Green, Cyan Purple, Black. This example uses file indirection to get data needed to define the tint transforms. Calculator environment variables can be used to define the XYZ of the media being printed on.

```
<CalculatorElement InputChannels="7" OutputChannels="3" InputNames=”Cyan Magenta Yellow Black Orange Green Purple”>
        <SubElements>
                <TintArrayElement Name="CyanScale" InputChannels="1" OutputChannels="3">
                        <float32NumberType>
                                <Data Filename="cyan-scaleXYZ.txt" Format="text"/>
                        </float32NumberType>
                </TintArrayElement>
                <TintArrayElement Name="CyanOffset" InputChannels="1" OutputChannels="3">
                        <float32NumberType>
                                <Data Filename="cyan-offsetXYZ.txt" Format="text"/>
                        </float32NumberType>
                </TintArrayElement>

                <TintArrayElement Name="MagentaScale" InputChannels="1" OutputChannels="3">
                        <float32NumberType>
                                <Data Filename="magenta-scaleXYZ.txt" Format="text"/>
                        </float32NumberType>
                </TintArrayElement>
                <TintArrayElement Name="MagentaOffset" InputChannels="1" OutputChannels="3">
                        <float32NumberType>
                                <Data Filename="magenta-offsetXYZ.txt" Format="text"/>
                        </float32NumberType>
                </TintArrayElement>

                <TintArrayElement Name="YellowScale" InputChannels="1" OutputChannels="3">
                        <float32NumberType>
                                <Data Filename="yellow-scaleXYZ.txt" Format="text"/>
                        </float32NumberType>
                </TintArrayElement>
                <TintArrayElement Name="YellowOffset" InputChannels="1" OutputChannels="3">
                        <float32NumberType>
                                <Data Filename="yellow-offsetXYZ.txt" Format="text"/>
                        </float32NumberType>
                </TintArrayElement>

                <TintArrayElement Name="BlackScale" InputChannels="1" OutputChannels="3">
                        <float32NumberType>
                                <Data Filename="black-scaleXYZ.txt" Format="text"/>
                        </float32NumberType>
                </TintArrayElement>
                <TintArrayElement Name="BlackOffset" InputChannels="1" OutputChannels="3">
                        <float32NumberType>
                                <Data Filename="black-offsetXYZ.txt" Format="text"/>
                        </float32NumberType>
                </TintArrayElement>

                <TintArrayElement Name="GreenScale" InputChannels="1" OutputChannels="3">
                        <float32NumberType>
                                <Data Filename="green-scaleXYZ.txt" Format="text"/>
                        </float32NumberType>
                </TintArrayElement>
                <TintArrayElement Name="GreenOffset" InputChannels="1" OutputChannels="3">
                        <float32NumberType>
```

```
                        <Data Filename="green-offsetXYZ.txt" Format="text"/>
                    </float32NumberType>
                </TintArrayElement>

                <TintArrayElement Name="OrangeScale" InputChannels="1" OutputChannels="3">
                    <float32NumberType>
                        <Data Filename="orange-scaleXYZ.txt" Format="text"/>
                    </float32NumberType>
                </TintArrayElement>
                <TintArrayElement Name="OrangeOffset" InputChannels="1" OutputChannels="3">
                    <float32NumberType>
                        <Data Filename="orange-offsetXYZ.txt" Format="text"/>
                    </float32NumberType>
                </TintArrayElement>

                <TintArrayElement Name="PurpleScale" InputChannels="1" OutputChannels="3">
                    <float32NumberType>
                        <Data Filename="purple-scaleXYZ.txt" Format="text"/>
                    </float32NumberType>
                </TintArrayElement>
                <TintArrayElement Name="PurpleOffset" InputChannels="1" OutputChannels="3">
                    <float32NumberType>
                        <Data Filename="purple-offsetXYZ.txt" Format="text"/>
                    </float32NumberType>
                </TintArrayElement>
        </SubElements>
        <MainFunction>
        {
                ;default background is D50 for relative white point
                env(bkgX) not if {pop 0.9642}
                env(bkgY) not if {pop 1.000000}
                env(bkgZ) not if {pop 0.8249}
                in{Yellow} 0 gt if {
                    in{Yellow} tint{YellowScale} mul(3)
                    in{Yellow} tint{YellowOffset} add(3)
                }
                in{Orange} 0 gt if {
                    in{Orange} tint{OrangeScale} mul(3)
                    in{Orange} tint{OrangeOffset} add(3)
                }
                in{Magenta} 0 gt if {
                    in{Magenta} tint{MagentaScale} mul(3)
                    in{Magenta} tint{MagentaOffset} add(3)
                }
                in{Green} 0 gt if {
                    in{Green} tint{GreenScale} mul(3)
                    in{Green} tint{GreenOffset} add(3)
                }
                in{Cyan} 0 gt if {
                    in{Cyan} tint{CyanScale} mul(3)
                    in{Cyan} tint{CyanOffset} add(3)
                }
                in{Purple} 0 gt if {
                    in{Purple} tint{PurpleScale} mul(3)
                    in{Purple} tint{PurpleOffset} add(3)
                }
                in{Black} 0 gt if {
                    in{Black} tint{BlackScale} mul(3)
                    in{Black} tint{BlackOffset} add(3)
                }
                out(0,3)
        }
        </MainFunction>
</CalculatorElement>
```