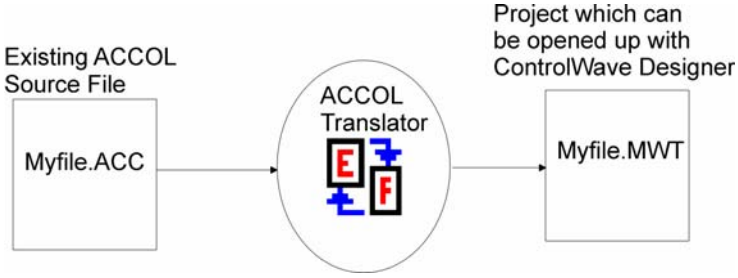
ACCOL Translator User's Guide

Table of Contents

What is the ACCOL Translator?

The ACCOL Translator is a Windows™ -based utility which translates an ACCOL II source (*.ACC) file into a ControlWave Designer project file (*.MWT). This translation process allows re-use of existing ACCOL II logic during development of new IEC 61131 projects for the ControlWave controller.



How the Translation Works – An Overview

The ACCOL Translator parses the existing ACC file, and generates a new project file from it. ACCOL II structures are converted to equivalent structures which can be used in ControlWave Designer with ACCOL III. At the completion of the translation, the new project is automatically opened within ControlWave Designer.

- Each ACCOL Task is converted to a program organization unit (POU). Each program POU is defined in the structured text (ST) language of IEC-61131-3.

- Each ACCOL signal is converted to an IEC 61131-3 variable. Logical and logical alarm signals become variables of type BOOL, and analog and analog alarm signals become variables of type REAL or UINT (unsigned INTEGER). String signals are converted to STRING variables.

- Any ACCOL II module *which has a counterpart in the ControlWave Designer ACCOL III firmware library* is converted to the equivalent function block in structured text (ST).

Does everything get translated?

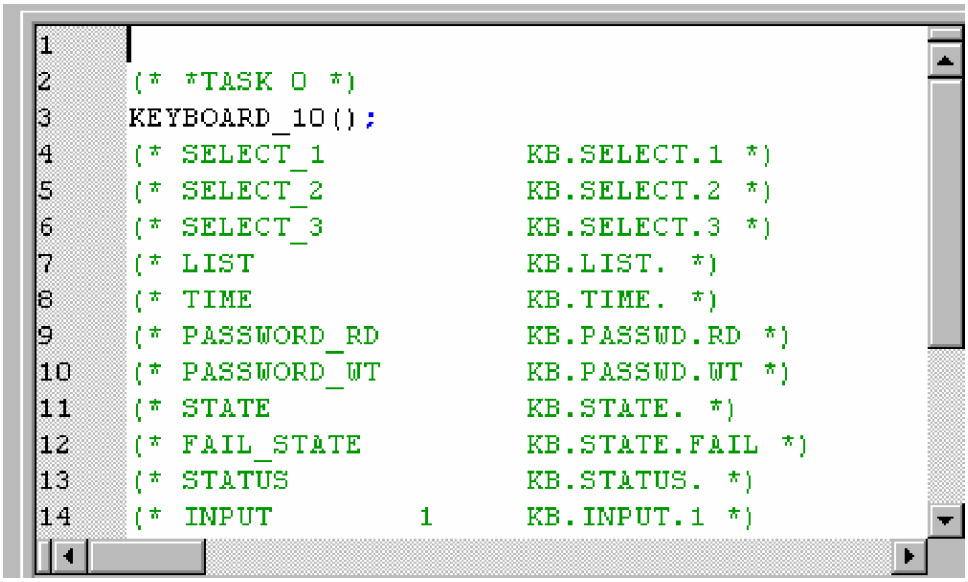
No. However, most ACCOL II modules are converted to IEC 61131-3 function blocks and included in the ACCOL III Firmware Library.

Often, the functionality of the untranslated ACCOL II modules, can be re-created by the user, if desired, using a combination of user-defined function blocks, and pre-defined ControlWave Designer functions, function blocks, and keywords.

Using the ACCOL Translator

What happens when the ACCOL Translator encounters an unsupported structure?

Any ACCOL II module or structure which is unsupported in IEC 61131-3 is converted to a comment, e.g. (* COMMENT *).



If an entire section of the ACCOL II source file cannot be converted, a WARNING message will be displayed in the output window of the ACCOL Translator.

Tips for Simplifying the Translation Process

Consider the Complexity of What You Are Trying to Translate

The ACCOL Translator is best for translating small ACCOL source files, or task-sized segments of them. Most users will obtain better results by stripping out modules that cannot be translated, and excess sections of the ACCOL source file prior to attempting the translation.

Complex files containing many unsupported modules tend to result in considerable work in the post-translation phase in order to remove large commented blocks of unusable code, as well as corrections of errors.

Reverse Compile Your ACCOL Load Prior to Translation

We recommend that after you edit the ACCOL source file in ACCOL Workbench, and you have compiled it, that you reverse compile it, prior to running the ACCOL Translator.

Rename Signals which are IEC 61131-3 Reserved Words Prior to Translation

NOTE: Some valid ACCOL II signal names may be invalid in IEC 61131-3 or may be converted to IEC 61131-3 keywords. You should rename these signals prior to translation. Particular examples of this include single character basename signals with no extension or attribute such as:

S..
R..

as well as other signal names such as:

DATE..
TIME..
CALC..

Some reserved words, such as ADD or SUB will automatically be changed by the ACCOL Translator to "ADDRWORD' or "SUBRWORD". When this is done, it may be necessary to modify any HMI packages that have been programmed to use the reserved words as variable names, especially if the "_USE_ACCOL_NAME" system variable has been turned on.

As of this writing, ACCOL Translator does not convert the words TRUE and FALSE. If these names are used for signals, the programmer will have to rename all usages of those names to #ON and #OFF, or to other names, as appropriate, in the ACCOL source file, prior to running the ACCOL Translator.

Numerical Values Must Begin with a Whole Number

If your code in ACCOL Workbench includes numerical values or calculator expressions that do NOT begin with a whole number to the left of the decimal point, for example,

.5
.732

etc., you must add the leading whole number, before performing the translation i.e.

0.5
0.732

or else your code will NOT be translated correctly.

Starting the ACCOL Translator

Click on **Start→ Programs → OpenBSI Tools → Utility Programs → ACCOL Translator**

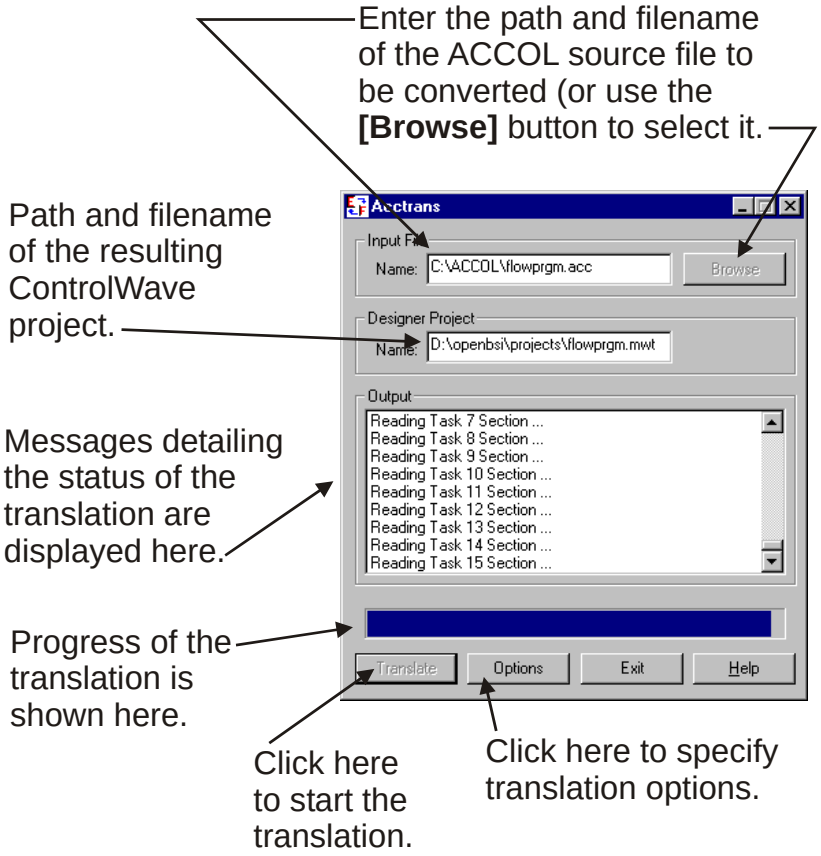
Starting the Translation

Type the path and filename of the ACCOL source file (*.ACC) to be translated into the **"Input File"** field, or use the **[Browse]** button to locate and specify the file. A name for the project will be chosen based on that name, with an extension of *.MWT. You can change the path and basename, but do not change the extension.

If desired, choose translation options by clicking on the **[Options]** button. (See *'Choosing Translation Options'* later in this section.)

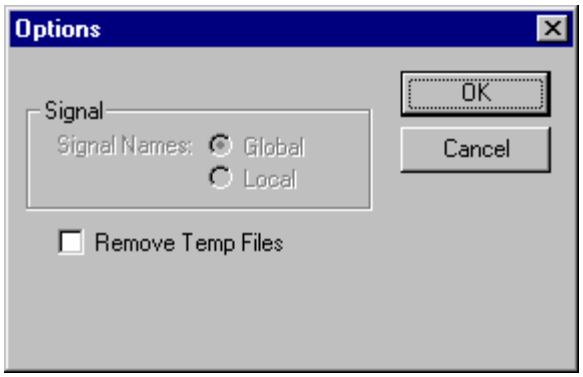
Finally, click on the **[Translate]** button to start the translation. The progress bar shows the amount of translation which is complete. Messages detailing the sections being translated will also be displayed.

ControlWave Designer will automatically be opened with the new project when the translation is complete. Click on **[Exit]** to close the ACCOL Translator.



Choosing Translation Options

Click on the **[Options]** button in the Acctrans dialog box to specify translation options:



Signal Names Reserved for future use.

Remove Temp Files Causes temporary files created in the ACCOL directory during the translation to be deleted after translation is complete. The default is to leave the temporary files.

Sections of the ACCOL Source File Which are NOT Translated

Several sections common to all ACCOL loads are NOT translated into IEC 61131-3. Typically, this is because the corresponding structures in IEC 61131-3 or the ControlWave controller are handled differently. The table, below, outlines the sections which are not translated.

ACCOL Section Name	Reason why the section is NOT translated into IEC 61131-3 structured text
*TARGET	The target controller, by definition, must support IEC 61131-3. The firmware libraries available determine which types of controllers can be used; not any specific identifier.
*SECURITY-CODES	Security is handled differently in the ControlWave controller; security is configured via the Flash Configuration Utility.
*MEMORY	Memory does NOT need to be explicitly allocated.
*BASENAMES	IEC 61131-3 variables do not use basename descriptive text.
*LOW-LEVEL	The Low-Level Analog Input board is supported for CW_10 and CW_30 only.
*FORMAT	Format codes are NOT supported in the ControlWave controller.
*COMMUNICATIONS	Communication ports are configured via the Flash Configuration Utility.

Translation of Data Arrays

Analog Data Array Declaration (both Read Only RO and Read Write RW)

Each analog data array is defined in the DATATYPE section as a new data type. The new data type consists of a one dimensional array of one dimensional arrays. This allows an equivalent set of row, column elements to be defined.

ACCOL Workbench (.ACC) File Structure	**Equivalent Construct in ControlWave Designer DATATYPE section**
*A-ARRAY *x* R*t* (*r*,*c*) where *x* = array number *r* = number of rows *c*= number of columns *t* = O – read only = W – read write Example: *A-ARRAY 1 RO (5,7)	TYPE ANA*x*_C : ARRAY[1..*c*] OF REAL; (*or* UINT) ANA*x*_R : ARRAY[1..*r*] OF ANA*x*_C; END_TYPE Where *x* = array number *r* = number of rows *c*= number of columns Equivalent Example: TYPE ANA1_C : ARRAY[1..7] OF REAL; ANA1_R : ARRAY[1..5] OF ANA1_C; END_TYPE

NOTE: The actual array variables (which are of the type defined in the DATATYPE section) appear in the Global_Variables section.

Analog Array Initialization (Read-Only Arrays)

Each read-only array is initialized in the system task called Inittsk. Inittsk runs once, on cold start.

NOTE: The same general syntax for initialization in ControlWave Designer is also valid for setting array values after initialization in read-write arrays.

ACCOL Workbench (.ACC) File Structure	**Equivalent Construct in ControlWave Designer**
*A-ARRAY *x* RO (*m*,*n*) *r1c1 r1c2 r1c3 …. r1cm* *r2c1 r2c2..r2c3… r2cm* *: : : :* *rnc1 rnc2 rnc3… rncm* where *x* = array number *m* = number of rows *n*= number of columns *r1c1 through rncm* = initial values Example: *A-ARRAY 1 RO (2,3) 17.5 3.4 8.2 8.3 7.3 4.1	(* Beginning of Array Section *) Ana_*x*_Array[*r1*] [*c1*] := *r1c1* Ana_*x*_Array[*r1*] [*c2*] := *r1c2* : : : : Ana_*x*_Array[*rn*] [*cm*] := *rncm* where *x* = array number *r* = number of rows *c*= number of columns *r1c1 through rncm* = initial values Equivalent Example: Ana_1_Array[1] [1] := 17.5000000; Ana_1_Array[1] [2] := 3.4000000; Ana_1_Array[1] [3] := 8.2000000; Ana_1_Array[2] [1] := 8.3000000; Ana_1_Array[2] [2] := 7.3000000; Ana_1_Array[2] [3] := 4.1000000;

Logical Data Array Declaration (both Read Only RO and Read Write RW)

Each logical data array is defined in the DATATYPE section as a new data type. The new data type consists of a one dimensional array of one dimensional arrays. This allows an equivalent set of row, column elements to be defined.

ACCOL Workbench (.ACC) File Structure	**Equivalent Construct in ControlWave Designer DATATYPE section**
*L-ARRAY *x* R*t* (*r*,*c*) where *x* = array number *r* = number of rows *c* = number of columns *t* = O – read only = W – read write Example: *L-ARRAY 1 RW (12,16)	TYPE LOG*x*_C : ARRAY[1..*c*] OF BOOL; LOG*x*_R : ARRAY[1..*r*] OF LOG*x*_C; END_TYPE Where *x* = array number *r* = number of rows *c* = number of columns Equivalent Example: TYPE LOG1_C : ARRAY[1..16] OF BOOL; LOG1_R : ARRAY[1..12] OF LOG1_C; END_TYPE

NOTE: The actual array variables (which are of the type defined in the DATATYPE section) appear in the Global_Variables section.

Logical Array Initialization (Read-Only Arrays)

Each read-write array is initialized in the system task called Inittsk. Inittsk runs once, on cold start.

NOTE: The same general syntax for initialization in ControlWave Designer is also valid for setting array values after initialization in read-write arrays.

ACCOL Workbench (.ACC) File Structure	**Equivalent Construct in ControlWave Designer**
*L-ARRAY *x* RO (*m*,*n*) *r1c1 r1c2 r1c3 …. r1cm* *r2c1 r2c2..r2c3… r2cm* *: : : :* *rnc1 rnc2 rnc3… rncm* where *x* = array number *m* = number of rows *n*= number of columns *r1c1 through rncm* = initial values <u>Example:</u> *L-ARRAY 1 RO (4,3) 1 0 0 1 1 0 0 0 1 1 0 1	(* Beginning of Array Section *) Log_*x*_Array[*r1*] [*c1*] := *r1c1* Log_*x*_Array[*r1*] [*c2*] := *r1c2* : : : : Log_*x*_Array[*rn*] [*cm*] := *rncm* where *x* = array number *r* = number of rows *c*= number of columns *r1c1 through rncm* = initial values <u>Equivalent Example:</u> Log_1_Array[1] [1] := TRUE; Log_1_Array[1] [2] := FALSE; Log_1_Array[1] [3] := FALSE; Log_1_Array[2] [1] := TRUE; Log_1_Array[2] [2] := TRUE; Log_1_Array[2] [3] := FALSE; Log_1_Array[3] [1] := FALSE; Log_1_Array[3] [2] := FALSE; Log_1_Array[3] [3] := TRUE; Log_1_Array[4] [1] := TRUE; Log_1_Array[4] [2] := FALSE; Log_1_Array[4] [3] := TRUE;

Translation of Signals

Signal Names

Signals are converted to IEC 61131-3 variables. IEC 61131-3 does NOT support periods embedded in variable names, therefore, the periods separating the basename and extension, and the extension and the attribute of an ACCOL signal name are converted to underscores.

ACCOL Workbench (.ACC) File Structure	**Equivalent Construct in ControlWave Designer**
basename.extension.attribute where: *basenam*e= up to 8 characters *extension* = up to 6 characters *attribute* = up to 4 characters Example: COMPRSR3.FLOW.TOT	*basename_extension_attribute* where: *basename*= up to 8 characters *extension* = up to 6 characters *attribute* = up to 4 characters Example: COMPRSR3_FLOW_TOT

NOTE: ControlWave Designer allows variable names up to 30 characters in length. If, however, you will be using a version of Open BSI Utilities *older than Version 4.1* to collect data, you should limit your variable names to no more than 20 characters.

NOTE: If you have an ACCOL II signal with a basename and attribute, but no extension, the extension will be replaced with the word ‘dot’. For example:

COMPRSR3..TOT will be translated to COMPRSR3_dot_TOT

Signal Definition

All ACCOL logical and logical alarm signals are converted to IEC 61131-3 variables of type BOOL. All ACCOL analog and analog alarm signals are converted to IEC 61131-3 variables of type REAL or UINT (unsigned Integer). By default, the variables are stored in the Global_Variables section of the project. When ACCOL signals are converted to IEC 61131-3 variables, certain characteristics of the signal are NOT translated, because they have no direct counterpart in IEC 61131-3.

The following characteristics of ACCOL signals are NOT translated to IEC 61131-3:

- Basename descriptive text (except in the case of alarms)
- Questionable data flag.
- *Initial* values for manual inhibit/enable, alarm inhibit/enable, control inhibit/enable flags. However, CALCULATOR statements making use of these flags are translated. See VAR-ATTRIB_GET, VAR_ATTRIB_SET, and VAR_CI_PROC.
- Units text for analog signals. Will be supported for analog alarms, however.
- ON/OFF text for logical signals (other than TRUE or FALSE) which are NOT alarms.

ACCOL Workbench (.ACC) File Structure	**Equivalent Construct in ControlWave Designer variable declaration section**
Logical Signals, Logical Alarm Signals *name type read_sec write_sec mi ci [ai] initial onoff [alarmtype] priority* where: *nam*e= signal name *type* = L for logical or LA for logical alarm *read_sec* = security level for read access *write_sec* = security level for write access *mi* = manual inhibit/enable flag *ci* = control inhibit/enable flag *[ai]* = alarm inhibit/enable flag (alarm signals only) *initial* = initial value of signal 0 = OFF 1 = ON. *onoff* = ON and OFF text *[alarmtype]* = condition which generates	Boolean Variables *name* **: BOOL :=** *initial;* where *name* = the variable name *initial* = the initial value of the variable

ACCOL Workbench (.ACC) File Structure	**Equivalent Construct in ControlWave Designer variable declaration section**
alarm (either TRUE or FALSE or CHANGE). (alarm signals only) *priority* = C – Critical (alarm signals only) N – Non-critical O – Operator Guide E – Event Example: PUMP1.START.CMD LA R1 W3 ME CE AE 0 RUN HALT TRUE C	 Example: PUMP1_START_CMD : BOOL := FALSE;

ACCOL Workbench (.ACC) File Structure	**Equivalent Construct in ControlWave Designer variable declaration section**
Analog Signals, Analog Alarm Signals *name type read_sec write_sec mi ci [ai] initial units [deadband] [limits] priority* where: *nam*e= signal name *type* = A for analog or AA for analog alarm *read_sec* = security level for read access *write_sec* = security level for write access *mi* = manual inhibit/enable flag *ci* = control inhibit/enable flag *[ai]* = alarm inhibit/enable flag (alarm signals only) *initial* = initial value of signal. *units* = units text e.g. HOURS *[deadband]* = alarm deadband value (alarm signals only) *[limits]* = alarm limit value(s) (alarms signals only) *priority* = C – Critical (alarm signals only) N – Non-critical O – Operator Guide	REAL Variables *name* **: REAL :=** *initial**;*** *- or-* *name* : **UINT** := *initial;* where *name* = the variable name *initial* = the initial value of the variable

ACCOL Workbench (.ACC) File Structure	Equivalent Construct in ControlWave Designer variable declaration section
E – Event Example: F101.FLOW.TOT AA R1 W3 ME CE AE 0.0000000 GPM HDB: 20 LDB: 5 HALM: 100 LALM: 10 C	 Example: F101_FLOW_TOT : REAL := 0.0000000;

System Signals

Many ACCOL II system signals do NOT have counterparts in ControlWave Designer with ACCOL III.

Certain similar variables, for example, for measuring time like #TIME.00*x* ACCOL signals, etc. are created in the SYS_VAR_WZ_DATA section of the Global_Variables, when you run the System Variable Wizard in ControlWave Designer. These system variables are distinguished from other variables by an underscore at the front of the variable name. For example, _TIME_005. A list of these appears below:

```
VAR_GLOBAL
     _TS_INHIB          AT %MX 3.0.3 : BOOL;
     _TS_REQ            AT %MX 3.0.2 : BOOL;
     _TIME_000          AT %MD 3.4 : DWORD;
     _TIME_001          AT %MD 3.8 : DINT;
     _TIME_002          AT %MW 3.12 : INT;
     _TIME_003          AT %MB 3.14 : SINT;
     _DAY_OF_WEEK       AT %MB 3.15 : SINT;
     _TIME_004          AT %MB 3.16 : SINT;
     _TIME_005          AT %MB 3.17 : SINT;
     _TIME_006          AT %MB 3.18 : SINT;
     _TIME_007          AT %MB 3.19 : SINT;
     _TIME_008          AT %MW 3.4 : INT;
     _CPU_OUT_ERR       AT %MD 3.24 : DINT;
```

```
    _T0_SLIP            AT %MD 3.1000 : DINT;
    _T1_SLIP            AT %MD 3.1032 : DINT;
    _T1_FPERR           AT %MD 3.1036 : DINT;
    _T2_SLIP            AT %MD 3.1064 : DINT;
    _T2_FPERR           AT %MD 3.1068 : DINT;
    _P1_TYPE            AT %MB 3.3000 : SINT;
    _P1_RECEIVES        AT %MD 3.3004 : DINT;
    _P1_TRANSMIT        AT %MD 3.3008 : DINT;
    _P1_POLLS           AT %MD 3.3012 : DINT;
END_VAR
```

Certain other ACCOL II system signals are translated to system variables with similar names in ACCOL III, and stored in the SYSTEM_VARIABLES group in the Global Variables worksheet (as opposed to the SYS_VAR_WIZ_DATA group.) If there are counterpart system signals with different names, the translated names based on the original name will be mapped to point to memory location for the actual system variable.

If there is no corresponding system variable, the translated variable will only be a variable, NOT a system variable.

They do NOT, *however,* have the same functionality of the ACCOL II equivalents; they are simply variables. Users can choose to rename them or otherwise change the logic they are used with.

Translation of Signal Lists

ACCOL signal lists are converted into lists in the **Inittsk** section of the project. The Inittsk section only executes *once*.

ACCOL Workbench (.ACC) File Structure	**Equivalent Construct in ControlWave Designer Structured Text (ST)**
***LIST** *listnumber* *listlinenum1 signal1* *listlinenum2 signal2* *listlinenum3 signal3* *: :* *listlinenumn signaln* where: *listnumber* = the number identifying this signal list *listlinenum1… n* = line numbers for the list *signal1 … signaln* = ACCOL signals <u>Example:</u> *LIST 5 10 PUMP1.RUN. 20 PUMP2.RUN. 30 PUMP3.RUN.	(* Beginning of Lists Section *) LIST_*size_id* (iiListnumber := *number*, ianyElement1 := *variable1*, ianyElement2 := *variable2*, ianyElement3 := *variable3*, *: :* ianyElement*n* := *variablen;* (* * INT * :=LIST_*id*.odiStatus; *) where: *size* = maximum number of items in the list. Can be 10, 20, 30, 50, or 100. *id* = a number identifying this list *number* = a number identifying the iilistnumber. *variable1…variablen* = the variable name <u>Example:</u> (* Beginning of Lists Section *) LIST_10_5 (iiListNumber := *number*, ianyElement1 := PUMP1_RUN, ianyElement2 := PUMP2_RUN, ianyElement3 := PUMP3_RUN); (* * INT * := LIST_10_5.odiStatus; *)

Translation of Process I/O

Because of direct access to I/O through variables, process I/O modules such as ANIN, DIGIN, DIGOUT are not converted.

The ANOUT module is converted, since it has a special usage with some function blocks, e.g. PID3TERM.

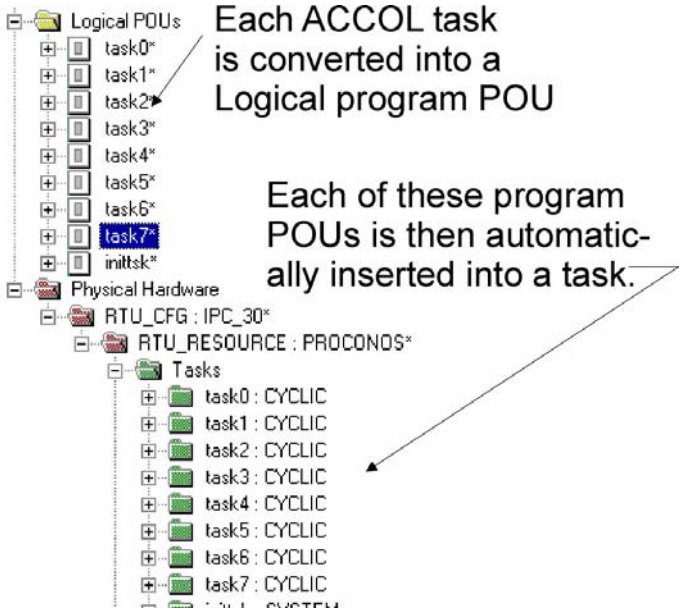
I/O points associated with process I/O boards are converted to global variables in the IO_GLOBAL_VARIABLES section of the Global_Variables page in the project tree.

Once you have opened the project in ControlWave Designer, you must make I/O assignments using the I/O Configurator (accessible in ControlWave Designer through **View→IO Configurator**).

Translation of Tasks, Modules, and Control Statements

Tasks

Each ACCOL task is converted into a logical program POU of the same name. For example, ACCOL Task 1, is converted to a logical POU called 'Task1'.This logical POU is then automatically inserted within an IEC 61131-3 task which handles the execution rate and priority.



Certain ACCOL task characteristics are converted into settings for the corresponding IEC 61131-3 task. For example, the ACCOL task rate becomes the **"Interval"** for the IEC 61131-3 cyclic task.

Task line numbers are automatically converted into (* Comment *) statements.

IMPORTANT

If there are 14 or fewer ACCOL tasks to be translated, the IEC 61131-3 task name will be the same as the newly created POU it contains. ControlWave Designer, however, can only have a maximum of 16 IEC 61131-3 tasks. Therefore, if your ACCOL source file contains *more* than 14 ACCOL tasks (including Task 0), all the ACCOL tasks will be converted to POUs, but those POUs will automatically be distributed among a maximum of 14 IEC 61131-3 tasks in ControlWave Designer. Two additional tasks (system tasks) are automatically created by the ACCOL Translator. The system tasks are used for initializing lists, and other special purposes.

For example, if your ACCOL source file contains 20 ACCOL tasks, the ACCOL Translator will generate 20 program POUs (1 for each of the ACCOL tasks), but only 16 IEC 61131-3 tasks (2 of which are System Tasks). The 20 POUs will be distributed among the 14 IEC 61131-3 non-system tasks. You can manually reassign the POUs, as desired, if you want them in different tasks. When deciding whether to reassign a particular POU, you need to consider whether another IEC 61131-3 task already has an identical execution rate and priority. You only need one task for a given rate / priority, since multiple program POUs can run in the same task.

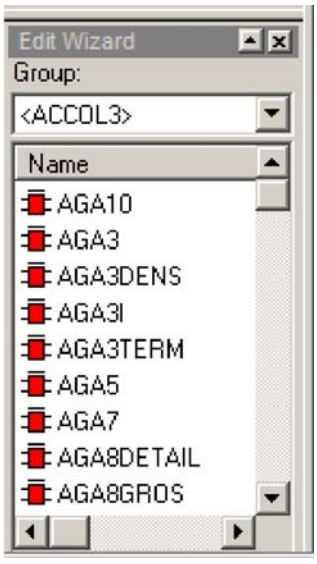
ACCOL Workbench (.ACC) File Structure	Equivalent Construct in ControlWave Designer
***TASK** *tasknumber* **RATE:** *rate* **PRI:** *priority* *linenum1 control_statement, comment or module* *linenum2 control_statement, comment or module* *linenum3 control_statement, comment or module* : : : *linenumn control_statement, comment or module* where: *tasknumber* = the number of the task *rate* = the execution rate of the task *priority* = the priority of the task *linenum1 linenumn* = task line numbers *control_statement* = ACCOL control statements *comment* = a comment *module* = ACCOL module Example: *TASK 1 RATE: 10.000000 PRI: 1 10 * IF (START.TASK<15) 20 * CALCULATOR 10 START.TASK=START.TASK+1 30 * ELSE 40 * C ** TWO DIMENSION FUNCTION MODULE 50 * C ** FUN.TEST1.HLT IS CTRL INHIBITED * 60 * IF (~FUN.TEST1.HLT) 70 * CALCULATOR 10 FUN1.TEST.CNT=FUN1.TEST.CNT+1 20 LINE.CNT.001=LINE.CNT.001+1 30 :IF(LINE.CNT.001>16) 40 LINE.CNT.001=1 50 :ENDIF 60 FUN.ROW.001=#ADATA 2[LINE.CNT.001] 70 FUN.COL.001=#ADATA 3[LINE.CNT.001] 80 * FUNCTION ARRAY FUN.ARRAY.001 ROW FUN.ROW.001 COLUMN FUN.COL.001	*linenum1* *control_statement, comment or fcnblock* *linenum2* *control_statement, comment or fcnblock* *linenum3* *control_statement, comment or fcnblock* : : : *linenumn* *control_statement, comment or fcnblock* where: *linenum1..n* = comment identifying the ACCOL task line number from the source ACC file *control_statement* = IEC 61131-3 control statements *comment* = a comment *module* = function block from Bbifsb library Example: *(* *TASK 1 RATE: 10.000000 PRI: 1 *)* *(* Line: 10 *)* IF (START_TASK<15.0) THEN *(* Line: 20 *)* START_TASK:=START_TASK+1.0; (* Line: 30 *) ELSE (* Line: 40 *) *(** TWO DIMENSION FUNCTION MODULE*)* *(* Line: 50 *)* *(* *FUN.TEST1.HLT IS CTRL INHIBITED *)* *(* Line: 60 *)* IF (NOT FUN_TEST1_HLT) THEN *(* Line: 70 *)* FUN1_TEST_CNT:=FUN1_TEST_CNT+1.0; LINE_CNT_001:=LINE_CNT_001+1.0; IF (LINE_CNT_001>16.0) THEN LINE_CNT_001:=1.0; END_IF; FUN_ROW_001:=Ana_2_Array[LINE_CNT_001];

OUTPUT FUN.OUT.001	FUN_COL_001:=Ana_3_Array[LINE_CNT_001]; *(* Line: 80 *)* *(*FUNCTION *)* FUNCTION_80(irFunctionArray := FUN_ARRAY_001, irRow := FUN_ROW_001, irColumn := FUN_COL_001); FUN_OUT_001 := FUNCTION_80.orOutput; *(* * REAL * := FUNCTION_80.orStatus;*)*

Control Statements

- IF, ENDIF, ELSE, ELSEIF are converted to IF, THEN, and ELSE statements in structured text.
- FOR statements are converted to FOR keywords in structured text.
- Most other ACCOL control statements (ABORT, GOTO, SUSPEND, RESUME, WAIT) are not supported, and so are converted into (* Comment *) statements.

Modules



- When parsing the ACCOL task, any ACCOL module encountered which exists in the ACCOL III library is declared as an IEC 61131-3 function block.
- Function block parameters are treated as assignment statements.
- The function blocks may have slightly different parameter names than the module terminals in the original module. Most parameter names are preceded by a prefix which identifies whether the parameter is an input variable to the function block, or an output variable from the function block, and also the variable type (e.g. BOOL, REAL, or UINT). For example, a module terminal named Setpoint, would become a function block parameter named irSetpoint.

The table, below, summarizes the meaning of the letters in the parameter name prefix.

Parameter Name Prefix	**Input or Output**	**Valid Data Types:**
ia, iany	INPUT	REAL, SINT, INT, DINT NOTE: You CANNOT use constants on parameters with the 'ia' or 'iany' prefix; only variables may be used.
iab	INPUT	BOOL variable - NO constants allowed.
iais	INPUT	STRING or INT variable. No constants.
iar	INPUT	REAL variable - NO constants allowed.
iarb	INPUT	REAL or BOOL variable. NO constants allowed.
iaus	INPUT	USINT variable or array of USINT. No constants.
ib	INPUT	BOOL variable or constant
idi	INPUT	DINT variable or constant
ii	INPUT	INT variable or constant
ioab	INPUT & OUTPUT	BOOL variable - NO constants allowed.
ioar	INPUT & OUTPUT	REAL variable - NO constants allowed.
ir	INPUT	REAL variable or constant
is, isi	INPUT	SINT variable or constant
is, istr	INPUT	STRING (must be surrounded by single quotes)
iudi	INPUT	UDINT variable or constant
iui	INPUT	UINT variable or constant
ius	INPUT	USINT variable or constant
ob	OUTPUT	BOOL variable or constant
odi	OUTPUT	DINT variable or constant
oi	OUTPUT	INT variable or constant
or	OUTPUT	REAL variable or constant
oud	OUTPUT	UDINT variable or constant
oui	OUTPUT	UINT variable or constant
ous	OUTPUT	USINT variable or constant

Syntax of ACCOL II Modules/Structures and Corresponding ACCOL III Structure

NOTE: In the table below, ACCOL II modules appear in the left column, and the corresponding ACCOL III translated fuction block or keyword in structured text (ST) is shown in the right column. For details on how the various function blocks work, please consult the online help files in ControlWave Designer.

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
10 * AAT FREQ_1 ANALOG.SIGNAL. FREQ_2 ANALOG.SIGNAL. COUNT_1 ANALOG.SIGNAL. COUNT_2 ANALOG.SIGNAL. LIST_1 ANALOG.SIGNAL. LIST_2 ANALOG.SIGNAL. LIST_3 ANALOG.SIGNAL. STATUS_1 ANALOG.SIGNAL. STATUS_2 ANALOG.SIGNAL. STATUS_3 ANALOG.SIGNAL. STATUS_4 ANALOG.SIGNAL.	**AUTOADJUST_10**((* iiOutListData := * INT * *)); (* FREQ_2 ANALOG.SIGNAL. *) iudiCntMain := REAL_TO_INT(ANALOG_SIGNAL), iudiCntSens := REAL_TO_INT(ANALOG_SIGNAL), iiInList := REAL_TO_INT(ANALOG_SIGNAL), iiOutListFactr := REAL_TO_INT(ANALOG_SIGNAL), (* LIST_3 ANALOG.SIGNAL. *)); ANALOG_SIGNAL := DINT_TO_REAL(AUTOADJUST_10.odiStatus1); ANALOG_SIGNAL := DINT_TO_REAL(AUTOADJUST_10.odiStatus2); ANALOG_SIGNAL := DINT_TO_REAL(AUTOADJUST_10.odiStatus3); ANALOG_SIGNAL := DINT_TO_REAL(AUTOADJUST_10.odiStatus4);
20 * AGA3 DIFF_PRESS ANALOG.SIGNAL. STAT_PRESS ANALOG.SIGNAL. ADJ_PRESS ANALOG.SIGNAL. ORIF_DIAM ANALOG.SIGNAL. PIPE_DIAM ANALOG.SIGNAL. ORIF_CONST ANALOG.SIGNAL. BASE_PRESS ANALOG.SIGNAL. BASE_TEMP ANALOG.SIGNAL. FLOW_TEMP ANALOG.SIGNAL. FPV_IN ANALOG.SIGNAL. POINT ANALOG.SIGNAL. SPEC_GRAV ANALOG.SIGNAL. TRACK ANALOG.OR.LOG OUTPUT ANALOG.SIGNAL.	**AGA3_20**(iarDiffPress := ANALOG_SIGNAL, iarStatPress := ANALOG_SIGNAL, iarAdjPress := ANALOG_SIGNAL, iarOrifDiam:= ANALOG_SIGNAL, iarPipeDiam := ANALOG_SIGNAL, iarOrifCon:= ANALOG_SIGNAL, iarBasePress := ANALOG_SIGNAL, iarBaseTemp := ANALOG_SIGNAL, iarFlowTemp := ANALOG_SIGNAL, iarFPV := ANALOG_SIGNAL, iiPoint := ANALOG_SIGNAL, iarSpecGrav := ANALOG_SIGNAL, iarbTrack := ANALOG_OR_LOG,); (* * DINT * := AGA3_20.odiStatus; *) ANALOG_SIGNAL := AGA3_20.orOutput;

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
30 * AGA3DENS DIFF_PRESS ANALOG.SIGNAL STAT_PRESS ANALOG.SIGNAL FLOW_TEMP ANALOG.SIGNAL TAP_LOC LOGICAL.SIGNAL ORIF_DIAM ANALOG.SIGNAL PIPE_DIAM ANALOG.SIGNAL ORIF_COEF ANALOG.SIGNAL PIPE_COEF ANALOG.SIGNAL ORIF_RTEMP ANALOG.SIGNAL PIPE_RTEMP ANALOG.SIGNAL VISCOSITY ANALOG.SIGNAL ISEN_COEF ANALOG.SIGNAL FLOW_DENS ANALOG.SIGNAL BASE_DENS ANALOG.SIGNAL REL_DENS ANALOG.SIGNAL TRACK LOGICAL.SIGNAL MASS_FLOW ANALOG.SIGNAL VOL_FLOW ANALOG.SIGNAL BASE_FLOW ANALOG.SIGNAL LIST ANALOG.SIGNAL	**AGA3DENS_30(** irDiffPress := ANALOG_SIGNAL, irStatPress := ANALOG_SIGNAL, iarFlowTemp := ANALOG_SIGNAL, ibTapLoc := LOGICAL_SIGNAL, irOrifDiam:= ANALOG_SIGNAL, irPipeDiam := ANALOG_SIGNAL, iarOrifCoef:= ANALOG_SIGNAL, iarPipeCoef := ANALOG_SIGNAL, iarOrifRefTemp:= ANALOG_SIGNAL, iarPipeRefTemp := ANALOG_SIGNAL, iarViscosity := ANALOG_SIGNAL, irIsenCoef := ANALOG_SIGNAL, irFlowDens := ANALOG_SIGNAL, iarBaseDens := ANALOG_SIGNAL, irRelDens := ANALOG_SIGNAL, irCutOff := LOGICAL_SIGNAL, iiList := REAL_TO_INT(ANALOG_SIGNAL)); ANALOG_SIGNAL := AGA3DENS_30.orMassFlow; ANALOG_SIGNAL := AGA3DENS_30.orVolFlow; ANALOG_SIGNAL := AGA3DENS_30.orBaseFlow;
40 * AGA3ITER DIFF_PRESS ANALOG.SIGNAL. STAT_PRESS ANALOG.SIGNAL. TAP_LOC ANALOG.SIGNAL. ADJ_PRESS ANALOG.SIGNAL. ORIF_DIAM ANALOG.SIGNAL. PIPE_DIAM ANALOG.SIGNAL. THERM_COEF1 ANALOG.SIGNAL. THERM_COEF2 ANALOG.SIGNAL. BASE_PRESS ANALOG.SIGNAL. BASE_TEMP ANALOG.SIGNAL. FLOW_TEMP ANALOG.SIGNAL. VISCOSITY ANALOG.SIGNAL. SPEC_GRAV ANALOG.SIGNAL. ISEN_COEF ANALOG.SIGNAL. Z_FLOWING ANALOG.SIGNAL. Z_BASE ANALOG.SIGNAL. POINT ANALOG.SIGNAL. TRACK ANALOG.OR.LOG OUTPUT ANALOG.SIGNAL. LIST ANALOG.SIGNAL. INPUT 1 ANALOG.SIGNAL.	**AGA3I_40(** irDiffPress := ANALOG_SIGNAL, iarStatPress := ANALOG_SIGNAL, irTapLoc := ANALOG_SIGNAL, iarAdjPress := ANALOG_SIGNAL, irOrifDiam:= ANALOG_SIGNAL, irPipeDiam := ANALOG_SIGNAL, iarOrifTCoef := ANALOG_SIGNAL, iarPipeTCoef := ANALOG_SIGNAL, iarBasePress := ANALOG_SIGNAL, iarBaseTemp := ANALOG_SIGNAL, iarFlowTemp := ANALOG_SIGNAL, iarViscosity := ANALOG_SIGNAL, iarSpecGrav := ANALOG_SIGNAL, iarIsenCoef := ANALOG_SIGNAL, iarZFlow := ANALOG_SIGNAL, iarZBase := ANALOG_SIGNAL, iarPoint := REAL_TO_INT(ANALOG_SIGNAL), iarbTrack := ANALOG_OR_LOG, (* * DINT * := AGA3I_40.odiStatus; *) iiOutList := REAL_TO_INT(ANALOG_SIGNAL), iarInput1 := ANALOG_SIGNAL,); ANALOG_SIGNAL := AGA3I_40.orOutput;.

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
50 * AGA3TERM DIFF_PRESS ANALOG.SIGNAL. STAT_PRESS ANALOG.SIGNAL. ADJ_PRESS ANALOG.SIGNAL. ORIF_DIAM ANALOG.SIGNAL. PIPE_DIAM ANALOG.SIGNAL. ORIF_CONST ANALOG.SIGNAL. BASE_PRESS ANALOG.SIGNAL. BASE_TEMP ANALOG.SIGNAL. FLOW_TEMP ANALOG.SIGNAL. FPV_IN ANALOG.SIGNAL. POINT ANALOG.SIGNAL. SPEC_GRAV ANALOG.SIGNAL. TRACK ANALOG.OR.LOG OUTPUT ANALOG.SIGNAL. LIST ANALOG.SIGNAL. INPUT 1 ANALOG.SIGNAL.	**AGA3TERM_50**(iarDiffPress := ANALOG_SIGNAL, iarStatPress := ANALOG_SIGNAL, iarAdjPress := ANALOG_SIGNAL, iarOrifDiam:= ANALOG_SIGNAL, iarPipeDiam := ANALOG_SIGNAL, iarOrifCon:= ANALOG_SIGNAL, iarBasePress := ANALOG_SIGNAL, iarBaseTemp := ANALOG_SIGNAL, iarFlowTemp := ANALOG_SIGNAL, iarFPV := ANALOG_SIGNAL, iiPoint := REAL_TO_INT(ANALOG_SIGNAL), iarSpecGrav := ANALOG_SIGNAL, iarbTrack := ANALOG_OR_LOG, (* iarCPrime := * ANY * *) (* iarFb := * ANY * *) (* iarFr := * ANY * *) (* iarY := * ANY * *) (* iarFpb := * ANY * *) (* iarFtb := * ANY * *) (* iarFtf := * ANY * *) (* iarFg := * ANY * *) (* iarExt := * ANY * *) (* * DINT * := AGA3TERM_50.odiStatus; *) iiOutList := REAL_TO_INT(ANALOG_SIGNAL),); (* INPUT 1 ANALOG.SIGNAL. *) ANALOG_SIGNAL := AGA3TERM_50.orOutput;
60 * AGA5 VOLUME ANALOG.SIGNAL. BASE_PRESS ANALOG.SIGNAL. BASE_TEMP ANALOG.SIGNAL. FPV_IN ANALOG.SIGNAL. SPEC_GRAV ANALOG.SIGNAL. VOL_%_CO2 ANALOG.SIGNAL. VOL_%_N2 ANALOG.SIGNAL. VOL_%_O2 ANALOG.SIGNAL. VOL_%_HE ANALOG.SIGNAL. VOL_%_CO ANALOG.SIGNAL. VOL_%_H2S ANALOG.SIGNAL. VOL_%_H2O ANALOG.SIGNAL. VOL_%_H2 ANALOG.SIGNAL. VOL_CONVERS ANALOG.SIGNAL. ENERGY_CONV ANALOG.SIGNAL. OUTPUT ANALOG.SIGNAL.	**AGA5_60**(iarVolume := ANALOG_SIGNAL, iarBasePress := ANALOG_SIGNAL, iarBaseTemp := ANALOG_SIGNAL, iarFPV := ANALOG_SIGNAL, iarSpecGrav := ANALOG_SIGNAL, irCO2 := ANALOG_SIGNAL, irN2 := ANALOG_SIGNAL, irO2 := ANALOG_SIGNAL, irHE := ANALOG_SIGNAL, irCO := ANALOG_SIGNAL, irH2S := ANALOG_SIGNAL, irH2O := ANALOG_SIGNAL, irH2 := ANALOG_SIGNAL, iarVolConvers := ANALOG_SIGNAL, iarEnergyConv := ANALOG_SIGNAL,); (* * DINT * := AGA5_60.odiStatus; *) ANALOG_SIGNAL := AGA5_60.orOutput;

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
70 * AGA7 FLOW_SWITCH LOGICAL.SIGNAL. DENS_SWITCH LOGICAL.SIGNAL. FLOW_TEMP ANALOG.SIGNAL. FLOW_PRESS ANALOG.SIGNAL. RATE ANALOG.SIGNAL. BASE_TEMP ANALOG.SIGNAL. BASE_PRESS ANALOG.SIGNAL. FPV_IN ANALOG.SIGNAL. ADJ_PRESS ANALOG.SIGNAL. FLOW_DENS ANALOG.SIGNAL. BASE_DENS ANALOG.SIGNAL. SPEC_GRAV ANALOG.SIGNAL. GRAV_TEMP ANALOG.SIGNAL. GRAV_PRESS ANALOG.SIGNAL. CALIB_FACTR ANALOG.SIGNAL. SPAN ANALOG.SIGNAL. OUTPUT ANALOG.SIGNAL.	**AGA7_70**(ibFlowSwitch := LOGICAL7SIGNAL, ibDensSwitch := LOGICAL_SIGNAL, iarFlowTemp := ANALOG_SIGNAL, iarFlowPress := ANALOG_SIGNAL, iarRate := ANALOG_SIGNAL, iarBaseTemp := ANALOG_SIGNAL, iarBasePress := ANALOG_SIGNAL, iarFPV := ANALOG_SIGNAL, iarAdjPress := ANALOG_SIGNAL, iarFlowDens := ANALOG_SIGNAL, iarBaseDens := ANALOG_SIGNAL, iarSpecGrav := ANALOG_SIGNAL, iarGravTemp := ANALOG_SIGNAL, iarGravPress := ANALOG_SIGNAL, iarCalibFactr := ANALOG_SIGNAL, iarSpan := ANALOG_SIGNAL,); (* * DINT * := AGA7_70.odiStatus; *) ANALOG_SIGNAL := AGA7_70.orOutput;
80 * AGA8DETAIL ENABLE LOGICAL.SIGNAL. PRIORITY ANALOG.SIGNAL. FLOW_TEMP ANALOG.SIGNAL. STAT_PRESS ANALOG.SIGNAL. BASE_TEMP ANALOG.SIGNAL. BASE_PRESS ANALOG.SIGNAL. LIST ANALOG.SIGNAL. ARRAY ANALOG.SIGNAL. COLUMN ANALOG.SIGNAL. ERROR ANALOG.SIGNAL. STATUS ANALOG.SIGNAL. Z_FLOWING ANALOG.SIGNAL. Z_BASE ANALOG.SIGNAL. FPV ANALOG.SIGNAL.	**AGA8DETAIL_80**((* ENABLE LOGICAL.SIGNAL. *) (* PRIORITY ANALOG.SIGNAL. *) iarFlowTemp := ANALOG_SIGNAL, iarStatPress := ANALOG_SIGNAL, iarBaseTemp := ANALOG_SIGNAL, iarBasePress := ANALOG_SIGNAL, iiList := REAL_TO_INT(ANALOG_SIGNAL), iarArray := ANALOG_SIGNAL, iiRowSel := * INT * *) iiRowSel := REAL_TO_INT(ANALOG_SIGNAL), (* ERROR ANALOG.SIGNAL. *)); ANALOG_SIGNAL := DINT_TO_REAL(AGA8DETAIL_80.odiStatus); ANALOG_SIGNAL := AGA8DETAIL_80.orZFlowing; ANALOG_SIGNAL := AGA8DETAIL_80.orZBase; ANALOG_SIGNAL := AGA8DETAIL_80.orFPV;
90 *AGA8GROSS ENABLE LOGICAL.SIGNAL. PRIORITY ANALOG.SIGNAL. FLOW_TEMP ANALOG.SIGNAL. STAT_PRESS ANALOG.SIGNAL. BASE_TEMP ANALOG.SIGNAL. BASE_PRESS ANALOG.SIGNAL. MODE ANALOG.SIGNAL. HEAT_VALUE ANALOG.SIGNAL. REF_T_HV ANALOG.SIGNAL. REF_P_HV ANALOG.SIGNAL. REL_DENS ANALOG.SIGNAL. REF_T_RD ANALOG.SIGNAL. REF_P_RD ANALOG.SIGNAL. MOLE_%_N2 ANALOG.SIGNAL. MOLE_%_CO2 ANALOG.SIGNAL. MOLE_%_H2 ANALOG.SIGNAL. MOLE_%_CO ANALOG.SIGNAL. ERROR ANALOG.SIGNAL. STATUS ANALOG.SIGNAL. Z_FLOWING ANALOG.SIGNAL. Z_BASE ANALOG.SIGNAL. FPV ANALOG.SIGNAL.	**AGA8GROS_90**((* ENABLE LOGICAL.SIGNAL. *) (* PRIORITY ANALOG.SIGNAL. *) iarFlowTemp := ANALOG_SIGNAL, iarStatPress := ANALOG_SIGNAL, iarBaseTemp := ANALOG_SIGNAL, iarBasePress := ANALOG_SIGNAL, iusMode := REAL_TO_USINT(ANALOG_SIGNAL), iarHeatValue := ANALOG_SIGNAL, iarRefTHV := ANALOG_SIGNAL, iarRefPHV := ANALOG_SIGNAL, iarRelDens := ANALOG_SIGNAL, iarRefTRD := ANALOG_SIGNAL, iarRefPRD := ANALOG_SIGNAL, irMoleN2 := ANALOG_SIGNAL, irMolCO2 := ANALOG_SIGNAL, irMoleH2 := ANALOG_SIGNAL, irMoleCO := ANALOG_SIGNAL, (* ERROR ANALOG.SIGNAL. *)); ANALOG_SIGNAL := DINT_TO_REAL(AGA8GROS_90.odiStatus); ANALOG_SIGNAL := AGA8GROS_90.orZFlowing; ANALOG_SIGNAL := AGA8GROS_90.orZBase; ANALOG_SIGNAL := AGA8GROS_90.orFPV;

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword

100 *ANOUT
DEVICE ;DEVICE_ID
INITIAL ;CHANNEL
OUTPUT 1 ;ANALOG_SIGNAL_OR_VALUE
ZERO 1 ;ANALOG_SIGNAL_OR_VALUE
SPAN \ 1 ;ANALOG_SIGNAL_OR_VALUE
TRACK 1 ;LOGICAL_SIGNAL
RESET 1 ;ANALOG_SIGNAL
	_0_1_ZERO := ;ANALOG_SIGNAL_OR_VALUE;
	_0_1_SPAN := ;ANALOG_SIGNAL_OR_VALUE;
	ANOUT_100(
	irInput :=(*;ANALOG_SIGNAL_OR_VALUE*)_0.0_1.0,
	irZero := _0.0_1.0_ZERO,
	irSpan :=
	_0.0_1.0_SPAN(*;LOGICAL_SIGNAL*)(*;ANALOG_SIGNAL*));
	:= ANOUT_100.orOutput;
	:= ANOUT_100.obTrack;
	:= ANOUT_100.orOutput;

110 * ARC_STORE ARCHIVE ARCH.FILE.ID MODE ARCH.MODE. STATUS ARCH.STATUS. PARAMETER 1 ARCH.PARAM.	**ARCHIVE_110(** iiArchiveNumber := REAL_TO_INT(ARCH_FILE_ID) , isiMode := REAL_TO_SINT(ARCH_MODE)(* iiArchiveList := * INT * *) (* iiOutList := * INT * *) (* isiContractHour := * SINT * *) (* irwFactor1 := * REAL * *) (* irwFactor2 := * REAL * *) (* idiSequenceIndex := * DINT * *)); (* * UINT * := ARCHIVE_110.ouiNumRecords; *) (* * UINT * := ARCHIVE_110.ouiOldestRecord; *) (* * UINT * := ARCHIVE_110.ouiNewestRecord; *) (* PARAMETER 1 ARCH.PARAM. *) ARCH_STATUS := DINT_TO_REAL(ARCHIVE_110.odiStatus);
120 * AUDIT MODE AUDIT.MODE. LIST AUDIT.LIST. STATUS AUDIT.STATUS.	**AUDIT_120(** (* MODE AUDIT.MODE. *) (* ibDisable := * BOOL8 * *) iiAuditList := REAL_TO_INT(AUDIT_LIST)(* ianyEventVar := * ANY * *)); (* * UINT * := AUDIT_120.ouiNumEvents; *) (* * UINT * := AUDIT_120.ouiOldestEvent; *) (* * UINT * := AUDIT_120.ouiNewestEvent; *) (* * UINT * := AUDIT_120.ouiNumAlarms; *) (* * UINT * := AUDIT_120.ouiOldestAlarm; *) (* * UINT * := AUDIT_120.ouiNewestAlarm; *) AUDIT_STATUS := DINT_TO_REAL(AUDIT_120.odiStatus);
130 * AVERAGER INPUT ANALOG.OR.LOG RESET LOGICAL.SIGNAL. TRACK LOGICAL.SIGNAL. SPAN ANALOG.SIGNAL. OUTPUT_1 ANALOG.SIGNAL. OUTPUT_2 ANALOG.SIGNAL. TIME ANALOG.SIGNAL.	**AVERAGER_130(** IarInput := ANALOG_OR_LOG, IbReset := LOGICAL_SIGNAL, IbTrack := LOGICAL_SIGNAL, IrSpan := ANALOG_SIGNAL,); ANALOG_SIGNAL := AVERAGER_130.orOutput_1; ANALOG_SIGNAL := AVERAGER_130.orOutput_2; ANALOG_SIGNAL := AVERAGER_130.orTime;

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
140 * CALCULATOR *calculator expression(s)*	*calculator expression(s)* *TRANSLATION NOTES:* • *Most calculator expressions are directly translated without difficulty to IEC 61131-3 expressions in structured text.* • *Two ACCOL III functions have also been added for calculation functions not present in IEC 61131-3: R_INT for truncating a REAL to an INT, and R_RND for rounding a REAL to the nearest INT.*
150 * COMMAND COMMAND LOGICAL.SIGNAL. OUTPUT LOGICAL.SIGNAL. DELAY ANALOG.SIGNAL. TRANSITION ANALOG.SIGNAL. ON_LIM_SW LOGICAL.SIGNAL. OFF_LIM_SW LOGICAL.SIGNAL. STATUS LOGICAL.SIGNAL. RUN_TIME ANALOG.SIGNAL. RESET LOGICAL.SIGNAL.	**COMMAND_150**(IbCommand := LOGICAL_SIGNAL, IrDelay := ANALOG_SIGNAL, IrTransition := ANALOG_SIGNAL, IbOn_Lim_Sw := LOGICAL_SIGNAL, IbOFF_Lim_Sw := LOGICAL_SIGNAL, IbReset := LOGICAL_SIGNAL,); LOGICAL_SIGNAL := COMMAND_150.obOutput; LOGICAL_SIGNAL := COMMAND_150.obStatus; ANALOG_SIGNAL := COMMAND_150.orRun_time;
160 * COMPARATOR MODE LOGICAL.SIGNAL. INPUT ANALOG.SIGNAL. SETPOINT ANALOG.SIGNAL. DEADBAND ANALOG.SIGNAL. OUTPUT_1 ANALOG.OR.LOG OUTPUT_2 ANALOG.OR.LOG OUTPUT_3 ANALOG.OR.LOG	**COMPARATOR_160**(IbMode := LOGICAL_SIGNAL, IrInput := ANALOG_SIGNAL, IrSetpoint := ANALOG_SIGNAL, IrDeadband := ANALOG_SIGNAL,); ANALOG_OR_LOG := COMPARATOR_160.orOutput_1; ANALOG_OR_LOG := COMPARATOR_160.orOutput_2; ANALOG_OR_LOG := COMPARATOR_160.orOutput_3; (* * BOOL * := COMPARATOR_160.obOutput_1;*) (* * BOOL * := COMPARATOR_160.obOutput_2;*) (* * BOOL * := COMPARATOR_160.obOutput_3;*)
170 * CUSTOM MODE CUSTOM.MODE. LIST CUSTOM.LIST. STATUS CUSTOM.STATUS.	**CUSTOM_170(** (* ioabInit := * ANY * *) (* idiRepeat := * DINT * *) iiMode := REAL_TO_INT(CUSTOM_MODE), iiCustomlist := REAL_TO_INT(CUSTOM_LIST)(* iiComPort := * INT * *) (* iiSlaveAddress := * INT * *) (* idiTimeout := * DINT * *) (* isIPAddress := * ANY * *)); (* * UDINT * := CUSTOM_170.oudDoneCount; *) (* * BOOL8 * := CUSTOM_170.obDoneFlag; *) CUSTOM_STATUS := DINT_TO_REAL(CUSTOM_170.odiStatus);

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
180 * DACCUMULATOR MODE ANALOG.SIGNAL. SCALE ANALOG.SIGNAL. INPUT_HIGH ANALOG.SIGNAL. INPUT_LOW ANALOG.SIGNAL. OUTPUT_HIGH ANALOG.SIGNAL. OUTPUT_LOW ANALOG.SIGNAL.	**DACCUMULATOR_180**(IoarMode := ANALOG_SIGNAL, IarScale := ANALOG_SIGNAL, IoarInput_High := ANALOG_SIGNAL, IoarInput_Low := ANALOG_SIGNAL,); ANALOG_SIGNAL := DACCUMULATOR_180.orOutput_High; ANALOG_SIGNAL := DACCUMULATOR_180.orOutput_Low;
190 * DEMUX INPUT ANY.SIGNAL. SELECT ANALOG.OR.LOG OUTLIST ANALOG.SIGNAL.	**DEMUX_190(** ianyInput := ANY_SIGNAL , iiSelect := REAL_TO_INT(ANALOG_OR_LOG), iiOutlist := REAL_TO_INT(ANALOG_SIGNAL)); (* * DINT * := DEMUX_190.odiStatus; *) ANALOG_SIGNAL := DEMUX_190.orOutput;
200 * DIFFERENTIATOR INPUT ANALOG.SIGNAL. RESET LOGICAL.SIGNAL. SPAN ANALOG.SIGNAL. OUTPUT ANALOG.SIGNAL.	**DIFFERENTIATOR_200**(irInput := ANALOG_SIGNAL, ibReset := LOGICAL_SIGNAL, irSpan := ANALOG_SIGNAL,); ANALOG_SIGNAL := DIFFERENTIATOR_200.orOutput;

210 * ENCODE
SELECT ANALOG.SIGNAL.
LIST ANALOG.SIGNAL.
ARRAY ANALOG.SIGNAL.
TYPE LOGICAL.SIGNAL.
MODE LOGICAL.SIGNAL.
INDEX ANALOG.SIGNAL.
STATUS ANALOG.SIGNAL.
INPUT 1 ANY.SIGNAL.
	ENCODE_210(
	isSelect := REAL_TO_SINT(ANALOG_SIGNAL),
	iiList := REAL_TO_INT(ANALOG_SIGNAL));
	(* ARRAY ANALOG.SIGNAL. *)
	(* TYPE LOGICAL.SIGNAL. *)
	(* MODE LOGICAL.SIGNAL. *)
	(* INDEX ANALOG.SIGNAL. *)
	(* * REAL * := ENCODE_210.orOutput1; *),
	irInput1 := ANALOG_SIGNAL);
	ANALOG_SIGNAL := DINT_TO_REAL(ENCODE_210.odiStatus);
	TRANSLATION NOTES:
	• *Only functions 3, 4, 5, and 6 are supported.*
	• *Julian Date/Time Stamp output for function 4 is sent to parameter Output1, instead of Input 1.*
	• *In function 6, missing values in the list will default to 0.*

220 * FOR *startat, endat, increment,* *action(s)* **230 * ENDFOR**	**FOR iLOOP_30 :=** *startat* **TO** *endat* **BY** *increment* **DO** *action(s)* **END_FOR;** *TRANSLATION NOTES:* • *This is a KEYWORD, not a function block.*

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
240 * FPV FLOW_TEMP ANALOG.SIGNAL. STAT_PRESS ANALOG.SIGNAL. SPEC_GRAV ANALOG.SIGNAL. CO2_MOLE ANALOG.SIGNAL. NMOLE ANALOG.SIGNAL. OUTPUT ANALOG.SIGNAL.	**FPV_240(** iarFlowTemp := ANALOG_SIGNAL, iarStatPress := ANALOG_SIGNAL, iarSpecGrav := ANALOG_SIGNAL, irCO2Mole := ANALOG_SIGNAL, irNMole := ANALOG_SIGNAL,); (* * DINT * := FPV_240.odiStatus; *) ANALOG_SIGNAL := FPV_240.orOutput;
250 * FUNCTION ARRAY ANALOG.SIGNAL. ROW ANALOG.SIGNAL. COLUMN ANALOG.SIGNAL. OUTPUT ANALOG.SIGNAL.	**FUNCTION_250(** iarFunctionArray := ANALOG_SIGNAL, irRow := ANALOG_SIGNAL, irColumn := ANALOG_SIGNAL,); (* * DINT * := FUNCTION_250.odiStatus; *) ANALOG_SIGNAL := FUNCTION_250.orOutput;
260 * HILOLIMITER INPUT ANALOG.SIGNAL. HIGH_LIMIT ANALOG.SIGNAL. LOW_LIMIT ANALOG.SIGNAL. OUTPUT_1 ANALOG.SIGNAL. OUTPUT_2 LOGICAL.SIGNAL. OUTPUT_3 LOGICAL.SIGNAL.	**HILOLIMITER_260(** irInput := ANALOG_SIGNAL, iarHighLimit := ANALOG_SIGNAL, iarLowLimit := ANALOG_SIGNAL,); ANALOG_SIGNAL := HILOLIMITER_260.orOutput_1; LOGICAL_SIGNAL := HILOLIMITER_260.obOutput_2; LOGICAL_SIGNAL := HILOLIMITER_260.obOutput_3; (* * BOOL * := HILOLIMITER_260.obError;*)
270 * HILOSELECT OUTPUT_1 ANALOG.SIGNAL. OUTPUT_2 ANALOG.SIGNAL. SELECT_1 ANALOG.OR.LOG SELECT_2 ANALOG.OR.LOG INLIST ANALOG.SIGNAL. INPUT 1 ANALOG.SIGNAL.	**HILOSELECT_270(** (* * DINT * := HILOSELECT_270.odiStatus; *) iiInlist := REAL_TO_INT(ANALOG_SIGNAL), (* INPUT 1 ANALOG.SIGNAL. *)); ANALOG_SIGNAL := HILOSELECT_270.orOutput_1; ANALOG_SIGNAL := HILOSELECT_270.orOutput_2; ANALOG := INT_TO_REAL(HILOSELECT_270.oiSelect_1); ANALOG := INT_TO_REAL(HILOSELECT_270.oiSelect_2); (* * REAL * := HILOSELECT_270.orStatus;*)
280 * HSCOUNT DEVICE 2 INITIAL 1 COUNT 1 ANALOG.SIGNAL. COUNT_ZERO 1 ANALOG.SIGNAL. COUNT_SPAN 1 ANALOG.SIGNAL. RESET 1 LOGICAL.SIGNAL. FREQUENCY 1 ANALOG.SIGNAL. FREQ_ZERO 1 ANALOG.SIGNAL. FREQ_SPAN 1 ANALOG.SIGNAL.	**HSCOUNT_280(** (* DEVICE 2 *) (* INITIAL 1 *) irCountZero := ANALOG_SIGNAL, irCountSpan := ANALOG_SIGNAL, ibReset := LOGICAL_SIGNAL, irFreqZero := ANALOG_SIGNAL, irFreqSpan := ANALOG_SIGNAL,); ANALOG_SIGNAL := HSCOUNT_280.orCount; ANALOG_SIGNAL := HSCOUNT_280.orFrequency;

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
290 * HWSTI DEVICE ;DEVICE_ID CHANNEL ;CHANNEL_ID COMMAND ANALOG.SIGNAL. DONE LOGICAL.SIGNAL. STATUS ANALOG.SIGNAL. PV ANALOG.SIGNAL. STIEU ANALOG.SIGNAL. SECVAR ANALOG.SIGNAL. MISMATCH LOGICAL.SIGNAL. CFGSTAT STRING.SIGNAL. SENSRTYP ANALOG.SIGNAL. DECONF ANALOG.SIGNAL. DAMPING ANALOG.SIGNAL. PVCHAR ANALOG.SIGNAL. CJTACT LOGICAL.SIGNAL. PIUOTDCF LOGICAL.SIGNAL. STITAG STRING.SIGNAL. FREQ6050 LOGICAL.SIGNAL. URV ANALOG.SIGNAL. LRV ANALOG.SIGNAL. URL ANALOG.SIGNAL. LRL ANALOG.SIGNAL. SERIALNO STRING.SIGNAL. STISWVER STRING.SIGNAL. SCRATCHPAD STRING.SIGNAL. XMITSTAT STRING.SIGNAL. COMERRS ANALOG.SIGNAL. POWERFAIL ANALOG.SIGNAL.	**HWSTI_290(** iiDevice := REAL_TO_INT(1.0), iiChannel := REAL_TO_INT(1.0), iiCommand := REAL_TO_INT(ANALOG_SIGNAL), ioaiStiEngUnits := ANALOG_SIGNAL, ioaiSensrTyp := ANALOG_SIGNAL, ioaiDeconf := ANALOG_SIGNAL, ioaiDamping := ANALOG_SIGNAL, ioaiPVChar := ANALOG_SIGNAL, ioabCJTAct := LOGICAL_SIGNAL, ioabPIUOTDCF := LOGICAL_SIGNAL, ioastrStiTag := STRING_SIGNAL, ioabFreq6050:= LOGICAL_SIGNAL, ioarURV:= ANALOG_SIGNAL, ioarLRV:= ANALOG_SIGNAL, ioarURL:= ANALOG_SIGNAL, iaiPwrFail := ANALOG_SIGNAL); LOGICAL_SIGNAL := HWSTI_290.obDoneFlag; ANALOG_SIGNAL := HWSTI_290.ostrCfgStat; ANALOG_SIGNAL := HWSTI_290.orPV; ANALOG_SIGNAL := HWSTI_290.orSecVar; LOGICAL_SIGNAL := HWSTI_290.obMismatch; ANALOG_SIGNAL := HWSTI_290.orLRL; STRING_SIGNAL := HWSTI_290.ostrSerialNo; STRING_SIGNAL := HWSTI_290.ostrSTISWVer; STRING_SIGNAL := HWSTI_290.ostrScratchPad; STRING_SIGNAL := HWSTI_290.ostrXmitStat; ANALOG_SIGNAL := HWSTI_290.ousCommErrs;
300 * IF (*condition***)** **310** ** executable statement(s) or module(s)* **320 * ENDIF**	**IF (***condition***) THEN** *executable statement(s), function(s), or function block(s)* **END_IF;** *TRANSLATION NOTES:* *This are KEYWORDs, not a function block*
330 * IF (*condition***)** **340** ** executable statement(s) or module(s)* **350 * ELSE** **360** ** executable statement(s) or module(s)* **370 * ENDIF**	**IF (***condition***) THEN** *executable statement(s), function(s), or function block(s)* **ELSE** *executable statement(s), function(s), or function block(s)* **END_IF;** *TRANSLATION NOTES:* *These are KEYWORDs, not a function block*

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
380 * IF (*condition1***)** **390** ** executable statement(s) or module(s)* **400 * ELSEIF (***condition2***)** **410** ** executable statement(s) or module(s)* **420 * ENDIF**	**IF (***condition1***) THEN** *executable statement(s), function(s), or function block(s)* **ELSEIF (***condition2***)** *executable statement(s), function(s), or function block(s)* **END_IF;** *TRANSLATION NOTES:* *These are KEYWORDs, not a function block*
430 * IF (*condition1***)** **440** ** executable statement(s) or module(s)* **450 * ELSEIF (***condition2***)** **460** ** executable statement(s) or module(s)* **470 * ELSE** **480** ** executable statement(s) or module(s)* **490 * ENDIF**	**IF (***condition1***) THEN** *executable statement(s), function(s), or function block(s)* **ELSEIF (***condition2***)** *executable statement(s), function(s), or function block(s)* **ELSE** *Executable statement(s), function(s), or function block(s)* **END_IF;** *TRANSLATION NOTES:* *These are KEYWORDs, not a function block*
500 * INTEGRATOR INPUT ANALOG.SIGNAL. RESET LOGICAL.SIGNAL. ZERO ANALOG.SIGNAL. SPAN ANALOG.SIGNAL. OUTPUT ANALOG.SIGNAL.	**INTEGRATOR_500**(irInput := ANALOG_SIGNAL, ibReset := LOGICAL_SIGNAL, irZero := ANALOG_SIGNAL, irSpan := ANALOG_SIGNAL,); ANALOG_SIGNAL := INTEGRATOR_500.orOutput;
510 * IP_CLIENT REMOTE STRING.SIGNAL. RESOLV_NAME ANALOG.SIGNAL. SERVR_ID ANALOG.SIGNAL. ACCESS_MODE ANALOG.SIGNAL. RESP_TMO ANALOG.SIGNAL. STRUCT_TYPE ANALOG.SIGNAL. SERVR_STRUCT_NO ANALOG.SIGNAL. SERVR_INDEX ANALOG.SIGNAL. ACCESS_TYPE ANALOG.SIGNAL. SERVR_SELECT ANALOG.SIGNAL. CLNT_STRCT_NO ANALOG.SIGNAL. CLNT_INDEX ANALOG.SIGNAL. CLNT_SELECT ANALOG.SIGNAL. CLNT_COUNT ANALOG.SIGNAL. STATUS_1 ANALOG.OR.LOG STATUS_2 ANALOG.SIGNAL.	**CLIENT_510**((* ioabInit := * ANY * *) (* iiProtocol := * INT * *) iaisRemoteNode := STRING_SIGNAL, (* RESOLV_NAME ANALOG.SIGNAL. *) iiServerID := REAL_TO_INT(ANALOG_SIGNAL), iiAccessMode := REAL_TO_INT(ANALOG_SIGNAL), idiTimeout := REAL_TO_DINT(ANALOG_SIGNAL), iiStructType := REAL_TO_INT(ANALOG_SIGNAL), iiServerStructNum := REAL_TO_INT(ANALOG_SIGNAL), iiServerStructIndex := REAL_TO_INT(ANALOG_SIGNAL), (* ACCESS_TYPE ANALOG.SIGNAL. *) iiServerStructSelect := REAL_TO_INT(ANALOG_SIGNAL), iiClientStructNum := REAL_TO_INT(ANALOG_SIGNAL), iiClientStructIndex := REAL_TO_INT(ANALOG_SIGNAL), iiClientStructSelect := REAL_TO_INT(ANALOG_SIGNAL), idiCount := REAL_TO_DINT(ANALOG_SIGNAL),); (* * BOOL8 * := CLIENT_510.obDoneFlag; *) (* * UDINT * := CLIENT_510.oudDoneCount; *) (* STATUS_2 ANALOG.SIGNAL. *) ANALOG _SIGNAL:= DINT_TO_REAL(CLIENT_510.odiStatus);

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
520 * IP_SERVER SERVR_ID ANALOG.SIGNAL. LIST_DB ANALOG.SIGNAL. AARRAY_DB ANALOG.SIGNAL. LARRAY_DB ANALOG.SIGNAL. ARCHIVE_DB ANALOG.SIGNAL. KNOWN_IP_NODES ANALOG.SIGNAL. RESOLV_NAME ANALOG.SIGNAL. STATUS_1 ANALOG.OR.LOG STATUS_2 ANALOG.SIGNAL.	**SERVER_520(** (* ioabInit := * ANY * *) (* iiProtocol := * INT * *) iiServerID := REAL_TO_INT(ANALOG_SIGNAL), iiListDB := REAL_TO_INT(ANALOG_SIGNAL), iiRealArrayDB := REAL_TO_INT(ANALOG_SIGNAL), iiBoolArrayDB := REAL_TO_INT(ANALOG_SIGNAL), (* iiAuthorizedNodes := * INT * *) (* iiStructType := * INT * *) (* iiClientStructNum := * INT * *) (* iiServerStructNum := * INT * *) (* ARCHIVE_DB ANALOG.SIGNAL. *) (* KNOWN_IP_NODES ANALOG.SIGNAL. *) (* RESOLV_NAME ANALOG.SIGNAL. *)); (* * BOOL8 * := SERVER_520.obDoneFlag; *) (* * UDINT * := SERVER_520.oudDoneCount; *) (* STATUS_2 ANALOG.SIGNAL. *) ANALOG_SIGNAL := DINT_TO_REAL(SERVER_520.odiStatus);
530 * ISO5167 DIFF_PRESS ANALOG.SIGNAL. STAT_PRESS ANALOG.SIGNAL. ADJ_PRESS ANALOG.SIGNAL. ORIF_DIAM ANALOG.SIGNAL. PIPE_DIAM ANALOG.SIGNAL. THERM_COEF1 ANALOG.SIGNAL. THERM_COEF2 ANALOG.SIGNAL. DEVICE ANALOG.SIGNAL. DEVICE2 ANALOG.SIGNAL. FLOW_TEMP ANALOG.SIGNAL. VISCOSITY ANALOG.SIGNAL. ISEN_COEF ANALOG.SIGNAL. DENSITY ANALOG.SIGNAL. BASE_DENS ANALOG.SIGNAL. STAT_P2 ANALOG.SIGNAL. POINT ANALOG.SIGNAL. TRACK ANALOG.OR.LOG OUTPUT ANALOG.SIGNAL. LIST ANALOG.SIGNAL.	**ISO5167_530**(iarDiffPress := ANALOG_SIGNAL, iarStatPress := ANALOG_SIGNAL, iarAdjPress := ANALOG_SIGNAL, iarOrifDiam:= ANALOG_SIGNAL, iarPipeDiam := ANALOG_SIGNAL, iarOrifCoef := ANALOG_SIGNAL, iarPipeCoef := ANALOG_SIGNAL, iiDevice := REAL_TO_INT(ANALOG_SIGNAL), iiDevice2 := REAL_TO_INT(ANALOG_SIGNAL), iarFlowTemp := ANALOG_SIGNAL, iarViscosity := ANALOG_SIGNAL, iarIsenCoef := ANALOG_SIGNAL, iarFlowDens := ANALOG_SIGNAL, iarBaseDens := ANALOG_SIGNAL, irStat_P2 := ANALOG_SIGNAL, iarScaleFact := REAL_TO_INT(ANALOG_SIGNAL), iarbTrack := LOGICAL_SIGNAL, (* * DINT * := ISO5167_530.odiStatus; *) iiOutList := REAL_TO_INT(ANALOG_SIGNAL),); ANALOG_SIGNAL := ISO5167_530.orOutput;

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
540 * LBBTI DEVICE ;DEVICE_ID CHANNEL ;CHANNEL MODE ANALOG.SIGNAL DGP ANALOG.SIGNAL DGPU ANALOG.SIGNAL DGPSUB ANALOG.SIGNAL SP ANALOG.SIGNAL SPU ANALOG.SIGNAL SPSUB ANALOG.SIGNAL RTDT ANALOG.SIGNAL RTDTU LOGICAL.SIGNAL RTDTSUB ANALOG.SIGNAL EST ANALOG.SIGNAL ESTU LOGICAL.SIGNAL ESTSUB ANALOG.SIGNAL TAG STRING.SIGNAL OUTPUT ANALOG.SIGNAL TRACK LOGICAL.SIGNAL ALARM LOGICAL.SIGNAL STATUS ANALOG.SIGNAL CFGSTAT STRING.SIGNAL ERRORCNT ANALOG_SIGNAL	**LBTI_540(** iiDevice := REAL_TO_INT(1.0), iiChannel := REAL_TO_INT(1.0), ioaiMode := ANALOG_SIGNAL, iorDGP := ANALOG_SIGNAL, iarDGPUnits := ANALOG_SIGNAL, iarDGPSUB := ANALOG_SIGNAL, iorSP := ANALOG_SIGNAL, iarSPUnits := ANALOG_SIGNAL, iarSPSUB := ANALOG_SIGNAL, iorRTDT := ANALOG_SIGNAL, iarRTDTUnits := LOGICAL_SIGNAL, iarRTDTSUB := ANALOG_SIGNAL, iorEST := ANALOG_SIGNAL, iarESTUnits := LOGICAL_SIGNAL, iarESTSUB := ANALOG_SIGNAL, iasTAG := STRING_SIGNAL, (*;ANALOG_SIGNAL_OR_VALUE*), iaiErrorCnt := ANALOG_SIGNAL); LOGICAL_SIGNAL := LBTI_540.obTrack; LOGICAL_SIGNAL := LBTI_540.obAlarm; ANALOG_SIGNAL := LBTI_540.odiStatus; STRING_SIGNAL_ := LBTI_540.ostrCfgStat;
550 * LEAD/LAG INPUT ANALOG.SIGNAL. DERIVATIVE ANALOG.SIGNAL. INTEGRAL ANALOG.SIGNAL. RESET LOGICAL.SIGNAL. OUTPUT ANALOG.SIGNAL.	**LEAD_LAG_550(** irInput := ANALOG_SIGNAL, irDerivative := ANALOG_SIGNAL, irIntegral := ANALOG_SIGNAL, ibReset := LOGICAL_SIGNAL,); ANALOG_SIGNAL := LEAD_LAG_550.orOutput;
560 * MASTER REMOTE ANALOG.SIGNAL. POINT ANALOG.SIGNAL. MODE ANALOG.SIGNAL. INTYPE ANALOG.SIGNAL. OUTTYPE ANALOG.SIGNAL. INDEX ANALOG.SIGNAL. INLIST ANALOG.SIGNAL. OUTLIST ANALOG.SIGNAL. STATUS_1 ANALOG.SIGNAL. STATUS_2 ANALOG.SIGNAL.	**MASTER_TO_CLIENT_560(** iaisRemoteNode := ANALOG_SIGNAL, iiServerID := REAL_TO_INT(ANALOG_SIGNAL), irAccessMode := ANALOG_SIGNAL(* INTYPE ANALOG.SIGNAL. *) (* OUTTYPE ANALOG.SIGNAL. *) , iiClientStructNum := REAL_TO_INT(ANALOG_SIGNAL), iiServerStructNum := REAL_TO_INT(ANALOG_SIGNAL)(* * BOOL8 * := MASTER_TO_CLIENT_560.obDoneFlag; *) (* * UDINT * := MASTER_TO_CLIENT_560.oudDoneCount; *)); ANALOG_SIGNAL := MASTER_TO_CLIENT_560.orStatus;
570 * MUX INLIST ANALOG.SIGNAL. SELECT ANALOG.OR.LOG OUTPUT ANY.SIGNAL.	**MUX_570(** iiInlist := REAL_TO_INT(ANALOG_SIGNAL), iiSelect := REAL_TO_INT(ANALOG_OR_LOG),); (* * DINT * := MUX_570.odiOutput; *) (* * DINT * := MUX_570.odiStatus; *) ANALOG_SIGNAL := MUX_570.orOutput; *TRANSLATION NOTES:* • *Signals in a list may be of type REAL, BOOL, or BYTE.* • *The irSelect terminal must be UINT.*

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
580 * PDO DEVICE 4 INITIAL 1 RESOLUTION 1 MODE 1 ANALOG.SIGNAL. OUTPUT 1 ANALOG.SIGNAL. ENABLE 1 LOGICAL.SIGNAL. MIN_TIME 1 ANALOG.SIGNAL. MAX_TIME 1 ANALOG.SIGNAL. SPAN 1 ANALOG.SIGNAL. INPUT 1 ANALOG.SIGNAL. HIGH_LIMIT 1 LOGICAL.SIGNAL. LOW_LIMIT 1 LOGICAL.SIGNAL. TRACK 1 LOGICAL.SIGNAL. RESET 1 ANALOG.SIGNAL.	**PDO_580**((* DEVICE 4 *) (* INITIAL 1 *) (* ioabRecalculate := * ANY * ,*) iiResolution := REAL_TO_INT(1.0), iiMode := REAL_TO_INT(ANALOG_SIGNAL), iarOutput:= ANALOG_SIGNAL, iabEnable := LOGICAL_SIGNAL, irMin_time := ANALOG_SIGNAL, irMax_time := ANALOG_SIGNAL, iarInput := ANALOG_SIGNAL, ibHigh_limit := LOGICAL_SIGNAL, ibLow_limit := LOGICAL_SIGNAL,); LOGICAL_SIGNAL := PDO_580.obTrack; ANALOG_SIGNAL := PDO_580.orReset; (* * BOOL * := PDO_580.obRaise_output;*) (* * BOOL * := PDO_580.obLower_output;*) (* * REAL * := PDO_580.orStatus;*)
590 * PID3TERM INPUT ANALOG.SIGNAL. SETPOINT ANALOG.SIGNAL. DEADBAND ANALOG.SIGNAL. PROPORTION ANALOG.SIGNAL. INTEGRAL ANALOG.SIGNAL. DERIVATIVE ANALOG.SIGNAL. RESET ANALOG.SIGNAL. TRACK LOGICAL.SIGNAL. OUTPUT ANALOG.SIGNAL. ERROR ANALOG.SIGNAL.	**PID3TERM_590**(irInput := ANALOG_SIGNAL, irSetpoint := ANALOG_SIGNAL, irDeadband := ANALOG_SIGNAL, irProportion := ANALOG_SIGNAL, irIntegral := ANALOG_SIGNAL, irDerivative := ANALOG_SIGNAL, irReset := ANALOG_SIGNAL, ibTrack := LOGICAL_SIGNAL,); ANALOG_SIGNAL := PID3TERM_590.orOutput; ANALOG_SIGNAL := PID3TERM_590.orError;
600 * SCHEDULER STROBE LOGICAL.SIGNAL. STATE LOGICAL.SIGNAL. RESET LOGICAL.SIGNAL. MODE ANALOG.SIGNAL. TRACK LOGICAL.SIGNAL. UNAVAILABLE 1 LOGICAL.SIGNAL. FAIL_STATE 1 LOGICAL.SIGNAL. RANK 1 ANALOG.SIGNAL. OUTPUT 1 LOGICAL.SIGNAL.	**SCHEDULER_600(** ibStrobe := LOGICAL_SIGNAL, ibState := LOGICAL_SIGNAL, ibReset := LOGICAL_SIGNAL, iiMode := REAL_TO_INT(ANALOG_SIGNAL)(* iaibDeviceInfo := * ANY * *)); (* * DINT * := SCHEDULER_600.odiStatus; *) (* UNAVAILABLE 1 LOGICAL.SIGNAL. *) (* FAIL_STATE 1 LOGICAL.SIGNAL. *) (* RANK 1 ANALOG.SIGNAL. *) 1.0 LOGICAL_SIGNAL := SCHEDULER_600.obTrack; ANALOG_SIGNAL := SCHEDULER_600.orOutput;
610 * SEQUENCER STROBE LOGICAL.SIGNAL. STATE ANALOG.SIGNAL. INPUT 1 LOGICAL.SIGNAL. OUTPUT 1 LOGICAL.SIGNAL.	**SEQUENCER_610**(ibStrobe := LOGICAL_SIGNAL, (* STATE ANALOG.SIGNAL. *) iiInputList := REAL_TO_INT(LOGICAL_SIGNAL), iiOutputlist := REAL_TO_INT(LOGICAL_SIGNAL),); (* * INT * := SEQUENCER_610.oiStatus; *) (* * DINT * := SEQUENCER_610.odiStatus; *)

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
620 * STEPPER STROBE LOGICAL.SIGNAL. HOLD_OFF LOGICAL.SIGNAL. DIRECTION LOGICAL.SIGNAL. INDEX ANALOG.SIGNAL. RESET LOGICAL.SIGNAL. RESET_INDEX ANALOG.SIGNAL. TRACK LOGICAL.SIGNAL. TRACK_INDEX ANALOG.SIGNAL. STEP ANALOG.SIGNAL. ARRAY ANALOG.SIGNAL. TIME ANALOG.SIGNAL. OUTPUT 1 ANALOG.OR.LOG	**STEPPER_620(** ibStrobe := LOGICAL_SIGNAL, ibHold_off := LOGICAL_SIGNAL, ibDirection := LOGICAL_SIGNAL, irIndex := ANALOG_SIGNAL, ibReset := LOGICAL_SIGNAL, irReset_index := ANALOG_SIGNAL, (* iiOutputList := * INT * *) irTrack_index := ANALOG_SIGNAL, iarbDataArray := ANALOG_SIGNAL),); (* OUTPUT 1 ANALOG.OR.LOG *) LOGICAL_SIGNAL := STEPPER_620.obTrack; ANALOG_SIGNAL := STEPPER_620.orStep; ANALOG_SIGNAL := STEPPER_620.orTime; (* * REAL * := STEPPER_620.orStatus;*)
630 * STORAGE RESET LOGICAL.SIGNAL. READ LOGICAL.SIGNAL. WRITE LOGICAL.SIGNAL. COLUMN LOGICAL.SIGNAL. INDEX ANALOG.SIGNAL. ARRAY ANALOG.SIGNAL. TYPE LOGICAL.SIGNAL. STATUS ANALOG.SIGNAL. LIST ANALOG.SIGNAL. INPUT 1 ANALOG.SIGNAL.	**STORAGE_630(** ibReset := LOGICAL_SIGNAL, ibRead := LOGICAL_SIGNAL, ibWrite := LOGICAL_SIGNAL, ibColumn := LOGICAL_SIGNAL, iuiIndex := REAL_TO_UINT(ANALOG_SIGNAL), iiArray := REAL_TO_INT(ANALOG_SIGNAL)(* TYPE LOGICAL.SIGNAL. *) , iiList := REAL_TO_INT(ANALOG_SIGNAL)); ANALOG_SIGNAL := DINT_TO_REAL(STORAGE_630.OdiStatus);
640 * TCHECK INLIST ;ANALOG_SIGNAL_OR_VALUE OUTLIST ;ANALOG_SIGNAL_OR_VALUE STATUS ;ANALOG_SIGNAL DGPSUB ;ANALOG_SIGNAL_OR_VALUE SPSUB ;ANALOG_SIGNAL_OR_VALUE RTDTSUB ;ANALOG_SIGNAL_OR_VALUE ESTSUB ;ANALOG_SIGNAL_OR_VALUE ERRORCNT ;ANALOG_SIGNAL_OR_VALUE	**TCHECK_640(** iiInlist := REAL_TO_INT((*;ANALOG_SIGNAL_OR_VALUE*)), iiOutlist := REAL_TO_INT((*;ANALOG_SIGNAL_OR_VALUE*))(*;ANALOG_SIGNAL*), iarDGPSUB := (*;ANALOG_SIGNAL_OR_VALUE*), iarSPSUB := (*;ANALOG_SIGNAL_OR_VALUE*), iarRTDTSUB := (*;ANALOG_SIGNAL_OR_VALUE*), iarESTSUB := (*;ANALOG_SIGNAL_OR_VALUE*), iaiErrorCnt := REAL_TO_INT((*;ANALOG_SIGNAL_OR_VALUE*))); := DINT_TO_REAL(TCHECK_640.OdiStatus);
650 * TIMER INPUT LOGICAL.SIGNAL. SETPOINT ANALOG.SIGNAL. RESET LOGICAL.SIGNAL. TIME ANALOG.SIGNAL. OUTPUT_1 LOGICAL.SIGNAL. OUTPUT_2 LOGICAL.SIGNAL.	**TIMER_650(** ibInput := LOGICAL_SIGNAL, irSetpoint := ANALOG_SIGNAL, ibReset := LOGICAL_SIGNAL); ANALOG_SIGNAL := TIMER_650.orTime; LOGICAL_SIGNAL := TIMER_650.obOutput_1; LOGICAL_SIGNAL := TIMER_650.obOutput_2;

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
660 * TOT/TRND INPUT ANALOG.OR.LOG START_HOUR ANALOG.SIGNAL. START_MIN ANALOG.SIGNAL. TIME ANALOG.SIGNAL. HOUR_SPAN ANALOG.SIGNAL. SHIFT_SPAN ANALOG.SIGNAL. DAY_SPAN ANALOG.SIGNAL. MONTH_SPAN ANALOG.SIGNAL. PREV_HOUR ANALOG.SIGNAL. PREV_SHIFT ANALOG.SIGNAL. PREV_DAY ANALOG.SIGNAL. PREV_MONTH ANALOG.SIGNAL. CUR_T_HOUR ANALOG.SIGNAL. CUR_T_SHIFT ANALOG.SIGNAL. CUR_T_DAY ANALOG.SIGNAL. CUR_T_MONTH ANALOG.SIGNAL. DERIVATIVE ANALOG.SIGNAL.	**TOT_TRND_660**(iarbInput := ANALOG_OR_LOG, irStart_Hour := ANALOG_SIGNAL, irStart_Min := ANALOG_SIGNAL, iarTime := ANALOG_SIGNAL, irHour_Span := ANALOG_SIGNAL, irShift_Span := ANALOG_SIGNAL, irDay_Span := ANALOG_SIGNAL, irMonth_Span := ANALOG_SIGNAL, (* irReset_Hour := * REAL * *) (* irReset_Shift := * REAL * *) (* irReset_Day := * REAL * *) (* irReset_Month := * REAL * *)); ANALOG_SIGNAL := TOT_TRND_660.orPrev_Hour; ANALOG_SIGNAL := TOT_TRND_660.orPrev_Shift; ANALOG_SIGNAL := TOT_TRND_660.orPrev_Day; ANALOG_SIGNAL := TOT_TRND_660.orPrev_Month; ANALOG_SIGNAL := TOT_TRND_660.orCur_T_Hour; ANALOG_SIGNAL := TOT_TRND_660.orCur_T_Shift; ANALOG_SIGNAL := TOT_TRND_660.orCur_T_Day; ANALOG_SIGNAL := TOT_TRND_660.orCur_T_Month; ANALOG_SIGNAL := TOT_TRND_660 .orDerivative;
670 * VLIMITER INPUT ANALOG.SIGNAL. TRACK LOGICAL.SIGNAL. RATE_UP ANALOG.SIGNAL. RATE_DOWN ANALOG.SIGNAL. OUTPUT_1 ANALOG.SIGNAL. OUTPUT_2 LOGICAL.SIGNAL.	**VLIMIT_670**(irInput := ANALOG_SIGNAL, ibTrack := LOGICAL_SIGNAL, irRateUp := ANALOG_SIGNAL, irRateDown := ANALOG_SIGNAL,); ANALOG_SIGNAL := VLIMIT_670.orOutput1; LOGICAL_SIGNAL := VLIMIT_670.obOutput2;
680 * VMUX TRACK LOGICAL.SIGNAL. RATE ANALOG.SIGNAL. OUTPUT ANALOG.SIGNAL. SELECT ANALOG.SIGNAL. INLIST ANALOG.SIGNAL. INPUT 1 ANALOG.SIGNAL	**VMUX_680**(ibTrack := LOGICAL_SIGNAL, irRate := ANALOG_SIGNAL, iiSelect := REAL_TO_INT(ANALOG_SIGNAL), irInlist := ANALOG_SIGNAL, irInput := ANALOG_SIGNAL); ANALOG_SIGNAL := VMUX_680.orOutput;

ACCOL II Module / Statement	ACCOL III Fuction Block / Keyword
690 * XMTR_INTERFACE DEVICE ;ANALOG_SIGNAL_OR_VALUE CHANNEL ;ANALOG_SIGNAL_OR_VALUE REMOTE ;ANALOG_SIGNAL_OR_VALUE MODE ;ANALOG_SIGNAL FORMAT ;ANALOG_SIGNAL_OR_VALUE ADDRESS ;ANALOG_SIGNAL_OR_VALUE COUNT ;ANALOG_SIGNAL_OR_VALUE LIST ;ANALOG_SIGNAL_OR_VALUE INDEX ;ANALOG_SIGNAL_OR_VALUE STATUS_1 ;ANALOG/LOGICAL_SIGNAL_OR_VALUE STATUS_2 ;ANALOG_SIGNAL	**XMTR_690(** iiDevice := REAL_TO_INT(1.0), iiChannel := REAL_TO_INT(2.0), iiChannel := REAL_TO_INT(ANALOG_SIGNAL_REM), ioaiMode := ANALOG_SIGNAL_MODE, iiFormat := REAL_TO_INT(ANALOG_SIGNAL_FRMT), iuiAddress := REAL_TO_UINT(ANALOG_SIGNAL_ADDR), iiCount := REAL_TO_INT(ANALOG_SIGNAL_CNT), iiList := REAL_TO_INT(ANALOG_SIGNAL_LIST), iiIndex := REAL_TO_INT(ANALOG_SIGNAL_INDX)); ANALOG_SIGNAL_STA1 := DINT_TO_REAL(XMTR_690.odiCommStatus); ANALOG_SIGNAL_STA2 := DINT_TO_REAL(XMTR_690.odiFnBlkStatus);

Application Notes on Module Translations:

The following are notes and tips for ControlWave programmers who are using ACCOL Translator to convert an ACCOL II load into a ControlWave project:

AGA8Detail and AGA8Gross – ENABLE and PRIORITY

When using the ACCOL translator to convert either the AGA8Detail or AGA8Gross module from ACCOL II to ControlWave Designer, two terminals from the ACCOL II modules do not exist in the corresponding ControlWave Designer function blocks: ENABLE and PRIORITY.

To emulate the ENABLE functionality, simply put the instance of the AGA8 function block into an IF statement, according to the following example:

```
IF (Enabled) THEN
        Enabled:=FALSE;
        AGA8DETAIL_1(
                iarFlowTemp:=Flow_Temp,
                iarStatPress:=Flow_Press,
                iarBaseTemp:=Base_Temp,
                iarBasePress:=Base_Press,
                iiList:=1);
        Status:=AGA8DETAIL_1.odiStatus;
        Z_Flow:=AGA8DETAIL_1.orZFlowing;
        Z_Base:=AGA8DETAIL_1.orZBase;
        FPV:=AGA8DETAIL_1.orFPV;
END_IF;
```

The purpose of the PRIORITY terminal in ACCOL II was used to LOWER the priority of the AGA8 calculation so that it did not interfere with other controller operations. Because of improved speed and performance in the ControlWave unit, this is no longer a consideration, so PRIORITY is unnecessary.

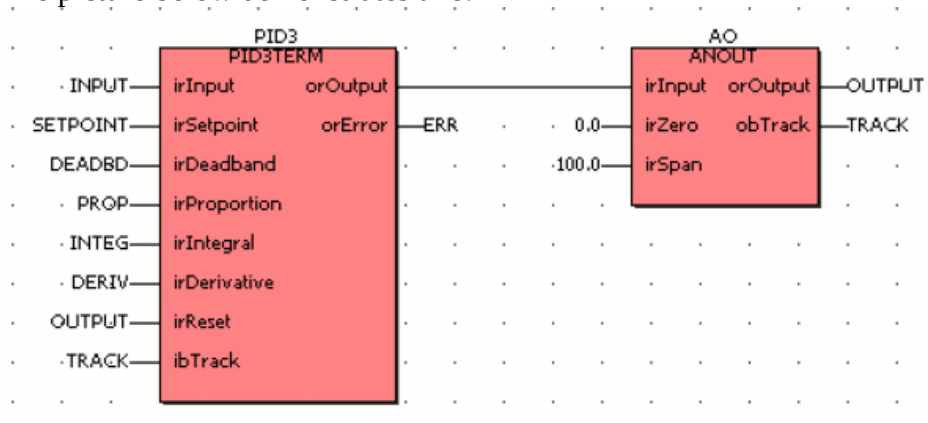
ANOUT Module – Implementing TRACK and RESET

Though ACCOL II required an ANOUT Module for analog outputs, because of the redesigned I/O system in ControlWave, the ANOUT function block is often unnecessary. REAL variables for output, zero, and span are wired directly to output pins.

If the TRACK and RESET terminals of the ACCOL II ANOUT module are used for control

purposes, however, (such as feedbacks to the TRACK and RESET terminals on a PID3TERM module), then an ANOUT function block must be configured in ControlWave Designer. The orOutput parameter of this block would be the variable wired to an I/O pin, and can also function as the RESET feedback to the PID3TERM function block, since it is the "clamped" value, based on the irZero and irSpan. The function block also has an obTrack parameter, which is functionally identical to the TRACK terminal from ACCOL.

The picture below demonstrates this:



Averager – Ensuring Proper Initial Value for TRACK

In ACCOL II, the TRACK Terminal in the AVERAGER Module had a default value of TRUE, meaning that if the terminal was unwired, the module is continuously active, with all outputs updating.

In ControlWave Designer, however, the AVERAGER function block's ibTrack parameter defaults to FALSE, meaning that the function block is disabled unless ibTrack is set to TRUE.

Because of this, if you translate an ACCOL load containing an AVERAGER module, and its TRACK terminal is unwired, the translated function block will not run. To correct this, you must insert the statement "ibTrack:=TRUE;" into the AVERAGER function block.

Comparator – Choosing Proper OUTPUT data type, Default for MODE

Because the COMPARATOR in ACCOL II could generate either an analog or a logical output, when the ACCOL translator converts it to a COMPARATOR function block, there will be two output parameters. The orOutput parameter will be datatype REAL and the obOutput will be datatype BOOL, and the same variable name is used for each parameter.

Because you cannot have the same variable in a function block defined with two different datatypes, you must delete the parameter with the datatype you don't want. For example, if you want the function block output to be type REAL, delete the obOutput parameter(s). If you want

the function block output to be type BOOL, delete the orOutput parameter(s).

Another difference between ACCOL II and ControlWave Designer relates to the mode. In ACCOL II, the MODE terminal defaults to ON. In ControlWave Designer, the ibMode parameter defaults to FALSE. If you want the translated COMPARATOR to operate the same way, you must change the initial value of the ibMode parameter to match that of the MODE terminal.

Custom Module

When using the ACCOL translator to convert a Custom module to its equivalent in ControlWave Designer, the programmer should be aware that certain terminals that did not exist in ACCOL II must be inserted after the translation is done.

As an example, the following code shows an ACCOL II Custom module intended for use as a Modbus Master (Mode 4):

```
 10 * CUSTOM
   MODE                    4.0000000
   LIST                    1.0000000
   STATUS          CUSTOM.STATUS.
```

The next code shows the above module, translated into a ControlWave Designer CUSTOM Function Block:

```
(* Line: 10 *)
(* CUSTOM Module *)

 CUSTOM_10(
(* ioabInit := * ANY * *)
(* idiRepeat := * DINT * *)
iiMode := REAL_TO_INT(4.0000000),
iiCustomlist := REAL_TO_INT(1.0000000)
(* iiComPort := * INT * *)
(* iiSlaveAddress := * INT * *)
(* idiTimeout := * DINT * *)
(* isIPAddress := * ANY * *));
(* * UDINT * := CUSTOM_10.oudDoneCount; *)
(* * BOOL8 * := CUSTOM_10.obDoneFlag; *)
CUSTOM_STATUS := DINT_TO_REAL(CUSTOM_10.odiStatus);
```

It should be pointed out that for a given Mode, not all of the parameters above may be needed. Also, the structure of the associated Custom List will change significantly. It is strongly suggested that the programmer check the ControlWave Designer online Help for the Mode being used to determine the correct list structure.

While there are many new parameters on this block that may require entries, one parameter that should be addressed is the first, “ioabInit”. The enhanced functionality of the CUSTOM function block requires that if any value on an input parameter is changed online, the block must be re-initialized to accept the new value. If the programmer chooses to accept this new option, he must create a new variable of type BOOL and change the line above to something like: “ioabInit := New_Bool_Var,”. If a change is made to an input parameter, the variable New_Bool_Var must be set TRUE to reset the Function Block to the new parameter.

Please remember that this is optional. If all input parameters are correct, and never need to be changed online, the block will initialize properly on startup without this parameter being configured.

Many other parameters on the CUSTOM function block, such as iiComPort, idiRepeat, iiSlaveAddress (or isIP_Address), and idiTimeout will have equivalent signals in the original ACCOL implementation. These parameters should be populated as described below:

Parameter Definitions:

iiComPort if communicating via serial lines, specifies the number of the serial port this function block will use (1=COM1, 2=COM2, etc.).

idiRepeat defines a minimum wait time (in milliseconds) between transmitted request messages when the function block is configured to transmit identical messages.

An idiRepeat value of 0 will cause the function block to transmit a request during the next execution after the obDoneFlag parameter is set to TRUE. The obDoneFlag parameter is set to TRUE during an execution when the reply is processed. An idiRepeat value of 0 is used for non-repeating requests.

Generally the idiRepeat parameter should be set equal to the cycle time of the task. This will allow requests to be transmitted and responses to be processed every cycle.

Setting the value greater than the task cycle time will hold FALSE the request transmission by the value of idiRepeat. However, the actual request transmission will not occur until the task execution that follows the elapse of the repeat time.

iiSlaveAddress If this function block serves as a Modbus Master, iiSlaveAddress specifies the Modbus address of the target slave. If this block is a Modbus Slave, iiSlaveAddress specifies the Modbus address of this function block. Valid addresses are 1 to 247. (Note: This is a change from the ACCOL implementation of the Modbus Slave, where the slave address was defined as

part of the port configuration.)

isIP_Address If IP communication is used (Open Modbus) this specifies the dotted decimal (eg. 147.200.75.3) address of the target RTU's IP port.

idiTimeout specifies the amount of time in milliseconds to wait for a response message from the slave. The value can range from 1 to 65534 milliseconds. When this parameter is not configured or is set to 0 the timeout value will default to 3 seconds. If a response is not received within 3 seconds after the message is transmitted then an error is reported.

Implementing Multiple Instances of CUSTOM Function Blocks

In ACCOL, when several CUSTOM modules had to execute, it was recommended that each module be followed by a WAIT FOR statement, which prevented the next CUSTOM module from executing until the first was finished. WAIT FOR statements are not available in Designer, nor are they necessary in this application. ControlWave Designer permits multiple "active" CUSTOM function blocks to be running. Simply put as many blocks as necessary in a program and let them run (be sure to use different instance names).

The messages will be queued and processed as quickly as possible. The programmer must be aware of the response time from the device being polled, to ensure that the idiTimeout parameter can be set properly. Also, another issue to watch is network loading, and keeping the communication line continuously busy.An important point to remember is that, if a CUSTOM function block is put into an iterative loop (FOR, WHILE, or REPEAT) to permit automatic changes of a parameter such as a target slave address, the "ioabInit" parameter MUST be manipulated as described below:

1. Execute the CUSTOM function block with ioabInit = TRUE. This will cancel any pending requests, and reset the CUSTOM function block with the new parameter values.

2. The CUSTOM function block will reset the ioabInit value to FALSE.

3. Execute the CUSTOM function block with ioabInit = FALSE until DONE = TRUE.

At this point, change the slave address or whatever else needs to change, and then return to step 1.

ENCODE Module – Functions 1, 2 ,9 and 10

If your ACCOL II load includes ENCODE Modules using Function 1 (Convert String Signal to Analog Values), Function 2 (Convert Analog Values to String Signal), Function 9 (Convert Logical Signal Sequences into a Decimal Value), or Function 10 (Convert a Decimal Value Into Logical Signal Sequences in Signal List), the programmer is encouraged to contact Bristol Technical Support for a library of function blocks to do these routines. The library 'Encode_Routines' requires installationof the 'Bit_UTIL' firmware library, which may be found in the OpenBSI libraries folder. Several array data types are included in the Encode_Routines library. See online help for more information on these libraries.

FUNCTION Module

When using the ACCOL translator to convert a FUNCTION module to its equivalent in ControlWave Designer, the programmer should be aware that array references from ACCOL II must be changed in ControlWave.

As an example, the following code shows an ACCOL FUNCTION module:

```
 10 * FUNCTION
   ARRAY                1.0000000
   ROW            P105.IN.
   COLUMN          T106.IN.
   OUTPUT         SPEC.GRAV.
```

The next code shows the above module, as it was translated from the ACCOL Translator:

```
FUNCTION_10(
iarFunctionArray := 1.0000000,
irRow := P105_IN,
irColumn := T106_IN(* * DINT * := FUNCTION_10.odiStatus; *)
);
SPEC_GRAV := FUNCTION_10.orOutput;
```

In the ACCOL Translator, each array is given its own variable type. The ACCOL array looked like this:

```
*A-ARRAY 1 RO (6,6)
(  1,1)      0.0000000     500.0000000     550.0000000     600.0000000
           700.0000000     900.0000000
(  2,1)    100.0000000    1100.0999756    1105.0000000    1110.4000244
          1116.8000488    1125.3000488
(  3,1)    125.0000000    1102.3000488    1106.0999756    1112.1999512
          1120.6999512    1128.0000000
```

```
( 4,1)    150.0000000    1106.5000000    1108.9000244    1113.0000000
          1122.5000000    1129.1999512
( 5,1)    200.0000000    1109.4000244    1112.6999512    1117.5999756
          1129.0999756    1134.3000488
( 6,1)    300.0000000    1112.0999756    1115.0999756    1119.0999756
          1133.3000488    1137.5000000
```

and the array type in Designer is:

```
TYPE
ANA1_C : ARRAY[1..6] OF REAL;
ANA1_R : ARRAY[1..6] OF ANA1_C;
END_TYPE
```

The resulting array variable is Ana_1_Array of type ANA1_R, declared as a PDD variable.

The code above must be modified to show the array variable, not the array name:

```
FUNCTION_10(
iarFunctionArray := Ana_1_Array,
irRow := P105_IN,
irColumn := T106_IN(* * DINT * := FUNCTION_10.odiStatus; *)
);
SPEC_GRAV := FUNCTION_10.orOutput;
```

All array declarations and value initializations are now done automatically in Translator.

Hand Held Terminal and LOGGER Module

When using the ACCOL translator to convert LOGGER modules and FORMAT expressions (used with the Termiflex Hand Held Terminal) from ACCOL II, the programmer must be made aware that there are no direct equivalents of these structures in ControlWave Designer. Therefore, LOGGER modules will be commented out by the ACCOL translator, and Formats simply will not appear.

In addition, the communication ports have no means of supplying power to the Termiflex.

Fortunately, there are numerous alternatives to the Hand Held Terminal functionality now available:

Data Viewer and PocketBSI:
These require no programming in ControlWave Designer and allow full access to all variables and arrays flagged as PDD (arrays must be registered via the REG_ARRAY function block). All that is required in ControlWave Configuration is an open Slave port. See the ControlWave

Designer online help for information on the REG_ARRAY function block.

ControlWave Keypad:
This is similar to the 33xx Keypad. It requires a configured DISPLAY function block and a variable list. Several versions are available, however the ControlWave Process Automation Controller and the ControlWave LP do not support this Keypad/Display. See the ControlWave Designer online help for information on the DISPLAY function block.

Maple Display OITs:
These use CUSTOM (Modbus) function blocks and variable lists. A sample project to facilitate programming is supplied with Open BSI.

If it is necessary to emulate the LOGGER module for report printing (eg. to a serial printer or HyperTerminal), several sample programs (using the Generic_Serial function block) are available from Bristol Technical Support.

LSCount Module Translation From ACCOL to Designer Function Block

ControlWave LP and CW_30 Digital Input board drivers generate counts for inputs if the point is configured properly in the I/O Configuration Wizard. The count value comes directly from the I/O point on the *_CNTR parameter.

However, the ControlWave, ControlWave Micro, ControlWave EFM, ControlWave GFC and the ControlWave XFC do not support interruptible digital inputs.

Therefore, when using the ACCOL translator to convert an LSCOUNT module to its equivalent in ControlWave Designer, The LSCOUNT module is not translated.

The following code shows an ACCOL LSCOUNT module using a Discrete Input point:

```
10 * LSCOUNT
        DEVICE                  1
        INITIAL                 1
        COUNT                   1     PULSE.COUNT.
        COUNT_ZERO              1     PULSE.COUNT.ZERO
        COUNT_SPAN              1     PULSE.COUNT.SPAN
        RESET           1       PULSE.COUNT.RST
        FREQUENCY               1     PULSE.RATE.
        FREQ_ZERO               1     PULSE.RATE.ZERO
        FREQ_SPAN       1       PULSE.RATE.SPAN
```

The next code shows the above module, as it was translated by the ACCOL Translator:

```
(* LSCOUNT_10(); *)
```

As you can see, there was no translation, because there is no equivalent function block in ControlWave Designer.

If using a ControlWave LP or a ControlWave_30, after translation, the user must go into the I/O Configuration Wizard and identify the Digital Input point that is used by the LSCOUNT module. The user must check the 'Enable Counter Processing' box. The variable representing the COUNT will be *_CNTR (where * is the Pin Name associated with the I/O point.) The user must then find the original ACCOL signal associated with the COUNT Terminal on the LSCOUNT Module (in our example, PULSE.COUNT) and substitute the *_CNTR variable name for this ACCOL signal name. The programmer must handle the Variable Type conversion (UDINT_TO_REAL).

If using any other type of ControlWave, a High Speed Counter input must be used. Once the High Speed Counter point is identified, the programmer must go through the same process of substituting the new variable for the ACCOL signal and performing Type Conversion.

The programmer may substitute an HSCOUNT module for the LSCOUNT module in ACCOL

```
HSCOUNT_10(
(* DEVICE 1 *)
(* INITIAL 1 *)
irCountZero := PULSE_COUNT_ZERO,
irCountSpan := PULSE_COUNT_SPAN,
ibReset := PULSE_COUNT_RST,
irFreqZero := PULSE_RATE_ZERO,
irFreqSpan := PULSE_RATE_SPAN);
PULSE_COUNT := HSCOUNT_10.orCount;
PULSE_RATE := HSCOUNT_10.orFrequency;
```

All the input parameters above are functionally identical to their ACCOL II equivalents. However, two more input parameters must be added:

iudiInput a variable of type UDINT representing the total pulses which have come in from a UDI card in ControlWave

ianyTime a variable of type DINT or UDINT representing the time in milliseconds to be used in the frequency calculation ("pulses since last execution"/ianyTime). If this parameter is unconfigured, the system elapsed time between block executions will be used.

Specifying the UDI Card in Software

The card's returned value is a count of pulses that have occurred since the last time the card was reset (a BOOL variable is available to force a reset upon download). This value is used for the

iudiInput terminal mentioned above.

When specifying the card, be sure to synchronize the card with the task that will contain the POU(s) that contain the HSCOUNT function blocks. If this is done, the count will always be fresh when the function block executes.

Modbus Communication with ControlWave Ethernet I/O

Many Network 3000 controllers are currently using Modbus, either serially or via Ethernet (Open Modbus) to exchange values with Bristol ControlWave Ethernet I/O blocks. While existing CUSTOM modules from ACCOL II will be converted using ACCOL Translator, the programmer may find it more convenient to use the same ControlWave Ethernet I/O blocks as direct I/O devices for the ControlWave.

To do this, you must set up IP communications to the Ethernet I/O blocks. If you had been using Serial RS485 communications to connect to the Ethernet I/O blocks, they must be re-configured for Ethernet.

The Ethernet I/O blocks must also be set up in I/O Configuration Wizard in ControlWave Designer. No function blocks or 'Custom' lists are necessary. It is strongly recommended that the I/O blocks be synchronized with a task.

Portstatus Modules

There is no direct translation of the Portstatus Module, however, ControlWave Designer programmers can achieve similar functionality through a combination of the PORTATTRIB function block and system variables.

PORTATTRIB can read and change characteristics (baud rate, etc.) for most port types; but like the Portstatus, it cannot change the port type. BSAP ports must remain at 8 bits, no parity, and 1 stop bit. Changes are written into the Flash configuration parameters, and remain permanent until changed again (power downs do NOT reset the port).

System variables may be used to collect statistics on port operation. See the online help in ControlWave Designer for information on system variables.
For information on the PORTATTRIB function block, please consult the online help in ControlWave Designer.

Slave and IP_Server Modules

When using the ACCOL translator to convert a Slave or IP_Server module to its equivalent in ControlWave Designer, the programmer should be aware that a parameter which did not exist as a terminal in ACCOL II, the “initialization” parameter, can be inserted after the translation is done.

As an example, the following code shows an ACCOL IP_Server module:

```
 10 * IP_SERVER
   SERVR_ID                  1.0000000
   LIST_DB                   1.0000000
   AARRAY_DB                 2.0000000
   LARRAY_DB                 3.0000000
   KNOWN_IP_NODES            4.0000000
   RESOLV_NAME               RESOLVE.SIG.
   STATUS_1                  SERVER.STAT.001
   STATUS_2                  SERVER.STAT.002
```

The next code shows the above module, translated into a ControlWave Designer SERVER Function Block:

```
(* Line: 10 *)
(* IP_SERVER Module *)

 SERVER_10(
(* ioabInit := * ANY * *)
(* iiProtocol := * INT * *)
iiServerID := REAL_TO_INT(1.0000000),
iiListDB := REAL_TO_INT(1.0000000),
iiRealArrayDB := REAL_TO_INT(2.0000000),
iiBoolArrayDB := REAL_TO_INT(3.0000000)
(* iiAuthorizedNodes := * INT * *)
(* iiStructType := * INT * *)
(* iiClientStructNum := * INT * *)
(* iiServerStructNum := * INT * *)
(* KNOWN_IP_NODES 4.0000000 *)
(* RESOLV_NAME RESOLVE.SIG.          *)
(* * BOOL8 * := SERVER_10.obDoneFlag; *)
(* * UDINT * := SERVER_10.oudDoneCount; *)
(* STATUS_2 SERVER.STAT.002        *)
);
SERVER_STAT_001 := DINT_TO_REAL(SERVER_10.odiStatus);
```

While there are many new parameters on this function block that may require entries, the parameter in question is the first, “ioabInit”. The enhanced functionality of the SERVER function block requires that if any value on an input parameter is changed online, the function block must be re-initialized to accept the new value. Programmers who choose to accept this new option must create a new variable of type BOOL and change the parameter line above to something like: “ioabInit := New_Bool_Var,”. If a change is made to an input parameter, the variable New_Bool_Var must be set TRUE to reset the function block to use the new value.

Please remember that this is optional. If all input parameters are correct, and never need to be changed online, the block will initialize properly on startup without this parameter being configured.

WAIT Statements in ControlWave Designer

When using the ACCOL Translator to convert tasks with WAIT statements into ControlWave Designer tasks, the programmer must be made aware that there are no equivalents to WAIT statements in the Structured Text language. However, it is possible to emulate WAIT statements if the Sequential Function Chart (SFC) language is used. It will be necessary to re-write the translated ACCOL task manually to SFC code.

First, each block of ACCOL code that was successfully translated must be copied in as an “action” in a Sequential Function Chart program, and given the qualifier “P” for Pulse, to make sure it only executes once (see examples).

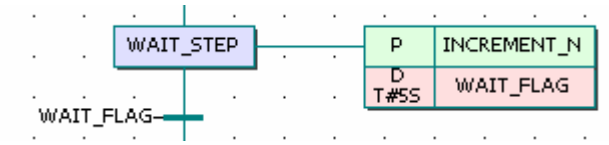
(In all subsequent discussion, it is assumed that the reader is familiar with SFC code.)

Translating WAIT DELAY Statements

Below is a segment of ACCOL code used to increment a counter and wait five seconds:

```
10 * CALCULATOR N=N+1
20 * WAIT DELAY 5 S
```

Here is one solution in SFC:



The action called INCREMENT_N has the code below:

```
IF (INCREMENT_N.X) THEN
        N:=N+1;
END_IF;
```

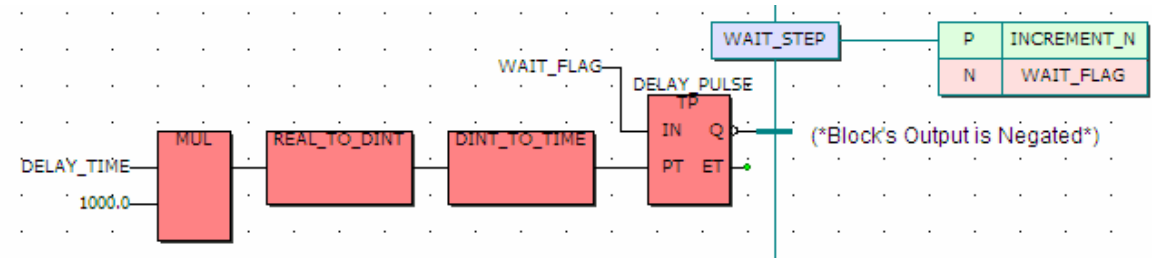
The action has also been given the qualifier “P” for pulse action, meaning it will execute once when the step is entered. However, the IF statement is necessary to prevent the action code from running again when the action is exited (INCREMENT_N.X is a flag that is driven FALSE when the action is inactive).

The second action is a variable action with a delay of five seconds. The BOOL variable “WAIT_FLAG” will be driven TRUE at that time, causing the transition to go TRUE as well, exiting the step.

It should be pointed out that the WAIT DELAY in ACCOL II used a signal or constant for the time delay, but the “D” qualifier requires a constant. If a variable wait time is desired, the ACCOL code would look something like this:

```
10 * CALCULATOR N=N+1
20 * WAIT DELAY DELAY.TIME S
```

The equivalent SFC code (based on the above example) would look something like this:



The Variable “DELAY_TIME” must be a variable of type REAL. Notice that the “WAIT_FLAG” action’s qualifier is now “N”, meaning that the flag is driven TRUE immediately. The TP function block called “DELAY_PULSE” will hold its negated output FALSE for the DELAY_TIME. It should also be noted that changing the value of DELAY_TIME while the TP block is timing up *will* affect the pulse length.

Translating WAIT FOR statements

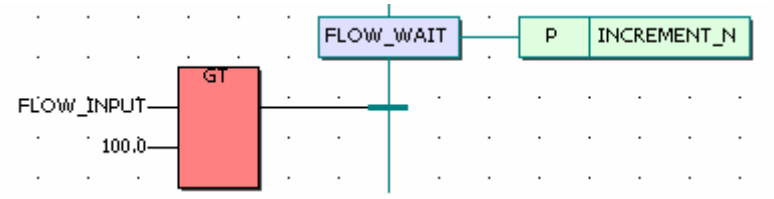
WAIT FOR statements have two scenarios: using a timeout, and no timeout. The following example shows ACCOL code with a WAIT FOR statement with no timeout:

```
10 * CALCULATOR N=N+1
20 * WAIT FOR (FLOW.INPUT > 100) .1, ,
```

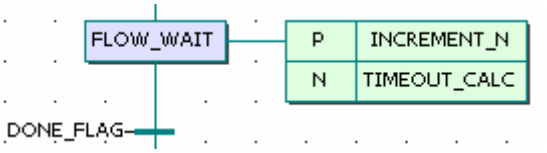
(Note: the value “.1” above is the execution frequency of the statement. In SFC, this is replaced

by the task execution rate.)

The equivalent SFC code would look something like this:



The next example incorporates a timeout, using an additional action:



The code for the action “TIMEOUT_CALC” is below:

```
IF (NOT(DONE_FLAG)) THEN
        TIMEOUT_FLAG:=FALSE; (*Resets Timeout Flag *)
        FLOW_FLAG:=FLOW_INPUT>100.0;
        IF (NOT(FLOW_FLAG)) THEN
                TIMEOUT_TIMER( (* TON Function Block *)
                        IN:=TRUE,
                        PT:=DINT_TO_TIME(REAL_TO_DINT(DELAY_TIME*1000.0)));
                TIMEOUT_FLAG:=TIMEOUT_TIMER.Q;
        END_IF;
        DONE_FLAG:=FLOW_FLAG OR TIMEOUT_FLAG;
ELSE
        TIMEOUT_TIMER(IN:=FALSE);  (* Resets TON block *)
        DONE_FLAG:=FALSE;
END_IF;
```

This action starts a TON function block, “TIMEOUT_TIMER”, and resets it, using the same instance name (that’s legal) when the step is exited. The flow condition flag is given priority over setting the timeout flag.

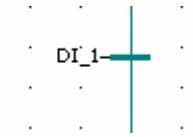
Translating WAIT DI, DIH, DIL statements

These statements require special consideration, because the original ACCOL II and Network 3000 products permitted “interrupt” points on discrete input cards (used by WAIT DIs), and, there are no interrupts on DI points in ControlWave.

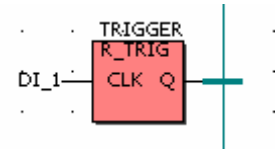
The programmer must understand that unlike WAIT DI statements in ACCOL, the transitions of Sequential Function Charts only execute at the task rate, and momentary contacts on ControlWave discrete inputs may be missed if they occur between task executions. It is therefore critical that:

1. Any discrete point used for a WAIT DI be a maintained contact, or, as an alternative, be a point on a High Speed Counter board
2. The board being used be synchronized to the task in question

The transition could be as simple as a single Boolean variable (possibly negated) from a discrete input pin:



or a rising/falling trigger function block (these may have to be initialized earlier in the code using the same instance name):



If you must wait for any change (rising or falling) on a DI, make a named transition:

with the code:

```
TRANS_1 := DI_1 XOR DI_OLD;
DI_OLD := DI_1;
```

Remember that with any of these, you are really waiting for a Boolean variable to change, NOT a DI!

Using a Counter card can pick up changes between task executions. Just change the transition code above to:

```
TRANS_1 := COUNT_1 <> COUNT_OLD;
COUNT_OLD := COUNT_1;
```

If you need timeouts on any of these, remember that they are essentially WAIT FOR statements, and use the timeout examples from the WAIT FOR section above.

A Final Warning on Transitions

The programmer should be reminded that ALL transitions in SFC are executed with each task execution. This could conceivably affect code execution, especially with WAIT DI emulations described above. If there is a possibility that an algorithm could be corrupted by a transition executing before the step is reached, the programmer has the option of moving the transition algorithm to an action, and having the algorithm drive a BOOL variable inside the action. This variable, then, would be the transition used to exit the step.

List of ACCOL II Modules / Structures NOT Translated

The table, below, shows all ACCOL II modules and control statements which have NOT been converted to IEC 61131-3 function blocks in the ACCOL III library and, in some cases, the reasons why. The ACCOL Translator will convert these modules to comments.

ACCOL Module / Structure	**Reason**
ABORT	Concept does not exist in IEC 61131-3
AGA8	Not available. Use AGA8Detail or AGA8Gross
ANIN / RANIN	Unnecessary. Direct access to I/O is provided.
BREAK	Not available.
CHARACTERIZE	Not available at this time.
CIM	Not available.
CNGMASTER	Not available.
CNGSLAVE	Not available.
DIGIN / RDIGIN	Unnecessary. Direct access to I/O is provided.
DIGOUT / RIDIGOUT	Unnecessary. Direct access to I/O is provided.
EASTATUS	Not available. Expanded Node Addresses utilized BSAP ports, and a different addressing scheme which does not apply here.
EMASTER	Not available.
EVP	Use EVP function block.
GBBTI	No hardware exists to support this option.
GOTO	Not available.
GPA8173	Use GPA8173 function block.
GSV	Use GSV function block.
HCBO	No hardware exists to support this option.
HSANIN	No hardware exists to support this option.
INTERNET PROTOCOL	Not available.
KEYBOARD	The DISPLAY function block performs a similar function.
LCBO	No hardware exists to support this option.
LIQUID_DENSITY	Use LIQUID_DENSITY function block.
LLANIN / RLLANIN	Platform dependent. Now supported for CW_10 and CW_30.
LSCOUNT / RLSCOUNT	Platform dependent.
NODESTATUS	Not available.
PDM / RPDM	Not available.
PORTSTATUS	Portattrib function block performs similar function.
RANOUT	Unnecessary. Direct access to I/O is provided.
RBE	RBE is now supported but cannot be translated.

ACCOL Module / Structure	Reason
REDUNDANCY	Implemented via REDUN_SWITCH function block.
RESUME	Concept does not exist in IEC 61131-3
RIOSTATS	Not available.
SLAVE	Not available. Consider using SERVER function block.
SMART	Not available.
SUSPEND	Concept does not exist in IEC 61131-3
SYS_3530	Unnecessary. Was limited to 3530 hardware only.
SYSTEM	Unnecessary. Was limited to Network 3000 hardware only.
SYSTEM0	Unnecessary. Was limited to Network 3000 hardware only.
TCOUNT	No hardware exists to support this option.
WAIT DELAY	See Application Notes.
WAIT DI/DIH/DIL RWAIT DI/DIH/DIL	See Application Notes.
WAIT FOR	See Application Notes.
WAIT TIME	Concept does not exist in IEC 61131-3
WATCHDOG	Not available.

Format of Initialization Files

Beginning with Open BSI Version 5.4, the ACCOL Translator generates a series of initialization files during its translation.

The information in these initialization files may be incorporated into the ControlWave project by configuring DB_LOAD and RBE function blocks, which read these files. You will have to add these function block(s) to the ControlWave project after you have finished the translation, and then configure them according to instructions in the DB_LOAD and RBE on-line help files in ControlWave Designer. The initialization files may be used to:

- Configure lists (Requires ControlWave firmware 04.40)
- Configure variables as alarms (NOT SUPPORTED BY FIRMWARE YET)
- Identify variables which should be collected via Report by Exception (Requires ControlWave firmware 04.40)
- Identify variables for audit collection (NOT SUPPORTED BY FIRMWARE YET)

Advanced users may want to edit the initialization files manually, with an ASCII text editor. Exercise extreme care when doing this, because syntactical errors could result in problems in your ControlWave project.

Do NOT leave spaces between lines of these files.

LISTS.INI

***LIST** *listnumber*
variable1
variable2
:
variablen

where *listnumber* is the number used to identify this list.

variable1-n are the variables in the list.

For example,

```
*LIST 1
@GV._AI_FOR_NON_ALARMS
@GV._ALARMS_BSAP_PORT1
@GV._ALARMS_BSAP_PORT1
@GV._ALARMS_BSAP_PORT10
@GV._ALARMS_BSAP_PORT11
```

@GV._ALARMS_BSAP_PORT11
@GV._T16_AVG_DUR

ALARMS.INI (NOT YET SUPPORTED IN FIRMWARE)

For each BOOL Variable:

Variable_ name
LOG *type priority*
UNITS *on_text / off_text*

where

variable_name	is the name of this BOOL variable
type	is an integer identifying the type of logical alarm 0 = Alarm on TRUE state 1 = Alarm on FALSE state 2 = Alarm on CHANGE of state
priority	is an integer identifying the alarm priority 0 = Critical 1 = Non-Critical 2 = Operator Guide 3 = Event
on_text / off_text	is the text to be displayed when the variable is TRUE or FALSE, respectively.

For each Analog Variable:

Variable_ name
HI *high_alarm_limit high_alarm_priority*
HIHI *high-high_alarm_limit high-high_alarm_priority*
LO *low_alarm_limit low_alarm_priority*
LOLO *low-low_alarm_limit low-low_alarm_priority*
HIDB *high_deadband*
LODB *low_deadband*
UNITS *engineering_units*

where

variable_name	is the name of this analog variable
high_alarm_limit	is the high alarm limit
high-high_alarm_limit	is the high high alarm limit
low_alarm_limit	is the low alarm limit
low-low_alarm_limit	is the low-low alarm limit
high_alarm_priority, *high-high_alarm_priority*, *low_alarm_priority*, *low-low_alarm_priority*	is an integer identifying the priority for each of these alarm limits. 0 = Critical 1 = Non-Critical 2 = Operator Guide 3 = Event
high_deadband	is the deadband to be applied to the high and high-high alarm limits. NOTE: This can be a entered as a constant value, or a variable name may be entered here, the value of which, will be used as the high deadband.
low_deadband	is the deadband to be applied to the low and low-low alarm limits. NOTE: This can be a entered as a constant value, or a variable name may be entered here, the value of which, will be used as the low deadband.
engineering_units	are the engineering units for this variable.

Examples:
@GV.PUMP1_STAT
LOG 2,1
UNITS RUN/STOP
@GV.TANK_LEVEL
HI 100 0
HIHI 200 1

LO 10 2
LOLO 5 3
HIDB 5
LODB 1
UNITS INCHES

AUDIT.INI (NOT YET SUPPORTED IN FIRMWARE)

variable1
variable2
:
variablen

where

variable1 – variablen are the names of variables you want marked for audit collection.

Example
@GV._AI_FOR_NON_ALARMS
@GV._ALARMS_BSAP_PORT1
@GV._ALARMS_BSAP_PORT10
@GV._ALARMS_BSAP_PORT11
@GV._ALARMS_BSAP_PORT2
@GV._ALARMS_BSAP_PORT3
@GV._ALARMS_BSAP_PORT4
@GV._ALARMS_BSAP_PORT5
@GV._ALARMS_BSAP_PORT6
@GV._ALARMS_BSAP_PORT7
@GV._ALARMS_BSAP_PORT8
@GV._ALARMS_BSAP_PORT9
@GV._ALARMS_IBP_DEST1
@GV._ALARMS_IBP_DEST2
@GV._ALARMS_IBP_DEST3
@GV._ALARMS_IBP_DEST4
@GV._ALARMS_PRESENT

RBE.INI

variable1 [deadband]
variable2 [deadband]
:
variablen [deadband]

where:

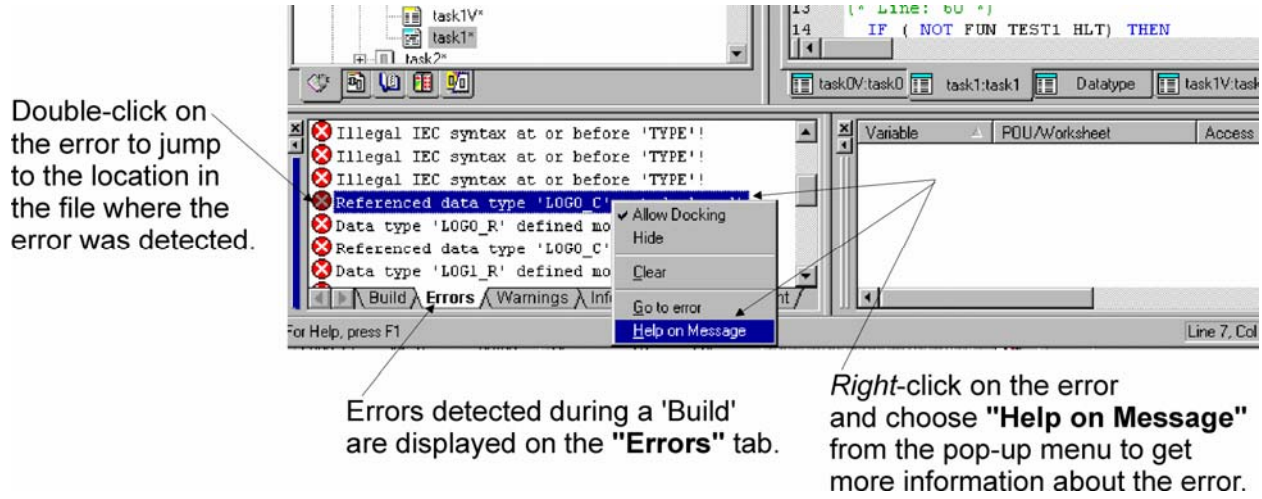
variable1 – variablen	are the names of variables you want marked for RBE collection.
[deadband]	is an RBE deadband applied to analog variables. This does NOT apply for BOOL variables. The deadband can be entered as a constant value, or a variable name can be entered, in which case, the value of that variable will serve as the deadband.

Example:
@GV.TANK_LEVEL @GV.TANK_LVL_DB
@GV.PRESSURE_NOW 10
@GV.PUMP_STARTED

Getting Help on Correcting Errors

Typically, after the ACCOL Translator has generated a ControlWave Designer project file, some number of errors will be present when a 'Make' command is issued in ControlWave Designer.

To locate the spot where the error was detected, go to the **"Errors"** tab in the lower left corner of the window, and double-click on the error message displayed. NOTE: Sometimes, the line on which an error was detected is not the exact location where the error was made.



To obtain a more detailed description of the error, *right*-click on the error message, and choose **"Help on Message"** from the pop-up menu.

Emerson Process Management
Remote Automation Solutions
1100 Buckingham Street
Watertown, CT 06795
Phone: +1 (860) 945-2262
Fax: +1 (860) 945-2525
www.EmersonProcess.com/Remote

Emerson Electric Canada, Ltd.
Bristol Canada
6338 Viscount Rd.
Mississauga, Ont. L4V 1H3
Canada
Phone: 905-362-0880
Fax: 905-362-0882
www.EmersonProcess.com/Remote

Emerson Process Management
BBI, S.A. de C.V.
Homero No. 1343, 3er Piso
Col. Morales Polanco
11540 Mexico, D.F.
Mexico
Phone: (52-55)-52-81-81-12
Fax: (52-55)-52-81-81-09
www.EmersonProcess.com/Remote

Emerson Process Management
Bristol Babcock, Ltd.
Blackpole Road
Worcester, WR3 8YB
United Kingdom
Phone: +44 1905 856950
Fax: +44 1905 856969
www.EmersonProcess.com/Remote

Emerson Process Management
Bristol, Inc.
22 Portofino Crescent,
Grand Canals Bunbury, Western Australia 6230
Mail to: PO Box 1987 (zip 6231)
Phone: +61 (8) 9725-2355
Fax: +61 (8) 8 9725-2955
www.EmersonProcess.com/Remote

NOTICE

"Remote Automation Solutions ("RAS"), division of Emerson Process Management shall not be liable for technical or editorial errors in this manual or omissions from this manual. RAS MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THIS MANUAL AND, IN NO EVENT SHALL RAS BE LIABLE FOR ANY INCIDENTAL, PUNITIVE, SPECIAL OR CONSEQUENTIAL DAMAGES INCLUDING, BUT NOT LIMITED TO, LOSS OF PRODUCTION, LOSS OF PROFITS, LOSS OF REVENUE OR USE AND COSTS INCURRED INCLUDING WITHOUT LIMITATION FOR CAPITAL, FUEL AND POWER, AND CLAIMS OF THIRD PARTIES.

Bristol, Inc., Bristol Babcock Ltd, Bristol Canada, BBI SA de CV and the Flow Computer Division are wholly owned subsidiaries of Emerson Electric Co. doing business as Remote Automation Solutions ("RAS"), a division of Emerson Process Management. FloBoss, ROCLINK, Bristol, Bristol Babcock, ControlWave, TeleFlow and Helicoid are trademarks of RAS. AMS, PlantWeb and the PlantWeb logo are marks of Emerson Electric Co. The Emerson logo is a trademark and service mark of the Emerson Electric Co. MULTIPROG wt is a registered trademark of KlÖpper und Weige Software GmbH.

All other trademarks are property of their respective owners.

The contents of this publication are presented for informational purposes only. While every effort has been made to ensure informational accuracy, they are not to be construed as warranties or guarantees, express or implied, regarding the products or services described herein or their use or applicability. RAS reserves the right to modify or improve the designs or specifications of such products at any time without notice. All sales are governed by RAS' terms and conditions which are available upon request.

EMERSON
Process Management

Return to the Table of Contents

Return to the List of Manuals