Honeywell

Control Language Application Module Overview

SW27-500

Honeywell

Implementation
Application Module - 3

Control Language Application Module Overview

SW27-500
Release 500
9/95

Copyright, Trademarks, and Notices

Revision 01 – September 1, 1995

About This Publication

This publication is an overview of Honeywell's TDC 3000^X Control Language for the Application Module (CL/AM). It tells you where and how to use CL/AM to augment the standard monitoring and control functions available in TDC 3000^X Systems. This publication should be read before using the *Control Language/Application Module Reference Manual* and the *Control Language/Application Module Data Entry* manual to implement CL/AM.

This overview does not cover all aspects of CL/AM in detail; rather, it gives you a good idea of what you have to consider as you implement CL/AM on your system.

This publication supports TDC 3000^X software release 500.

Table of Contents

1 INTRODUCTION

1.1 Purpose of This Manual
1.2 References

2 CL ENVIRONMENTS

3 CL ON APPLICATION MODULES

3.1 CL/AM Use
3.2 Database
3.3 Standard Capabilities
3.4 CL/AM Execution
3.5 CL/AM Program Example
3.5.1 Data Point for the Example Program
3.5.2 Program Statements
3.6 File Concepts
3.6.1 Program Source Entry
3.7 Compilation
3.8 Linking
3.9 Activating the Data Point
3.10 Summary of CL/AM Program Installation
3.11 Custom Data Segments
3.12 CL/AM Program Examples
3.12.1 Generic Package Example
3.12.2 Average Tank Temperature Example
3.12.3 Emergency Cooling Example
3.13 Parameter Lists
3.14 Data Types
3.14.1 Type Data-Point Identifier
3.14.2 Type Enumeration
3.14.3 Type String
3.15 CL/AM Block as PV or Control Algorithm

INTRODUCTION
Section 1

This section tells you what this publication is about.

1.1 PURPOSE OF THIS MANUAL

This manual is an introduction to the use of the Control Language for the Applications Module (CL/AM), a tool that enables users to extend the built-in capabilities of a TDC 3000^X control system. CL/AM is similar to general programming languages such as Fortran, BASIC or Pascal, but is designed to be particularly easy for control engineers to use on control applications.

This manual provides an introduction and general orientation for those who are new to the Control Language or to programming in general. It is not intended to cover all the details and nuances of the language.

The Control Language also can be used on Multifunction Controllers (CL/MC) and Process Managers (CL/PM) within a TDC 3000^X System. Section 2 provides some guidelines for deciding which of these CL environments is best suited for a particular application.

Section 3 explains the use of CL/AM, starting with a look at how the database is organized, then considers the standard functions that are available without CL/AM, and goes through a CL/AM programming example.

1.2 REFERENCES

The Control Language for the Applications Module is defined in the *Control Language/Application Module Reference Manual* in the *Implementation/Application Module - 3* binder.

Procedures for installing and for modifying CL/AM programs are in the *Control Language/Application Module Data Entry* manual, in the *Implementation/Application Module - 3* binder.

CL ENVIRONMENTS
Section 2

This section positions the Control Language within the functions available on TDC 3000X Systems.

Figure 2-1 shows a block diagram of an example TDC 3000X System, illustrating the positions of Application Modules (AMs), Multifunction Controllers (MCs), and Process Managers (PMs) in the TDC 3000X hierarchy. There is a considerable overlap between application functions that can be done in either an AM, an MC, or a PM, but there are also differences between the capabilities of these types of nodes that often dictates the use of one or the other:

1. The AM has a larger capacity for programs, data, and processing.

2. The AM has a wider access to data. Referring to Figure 2-1, an MC has access to the data in all the MCs connected to its C-link, which is a subset of the data that could be on one Data Hiway. A PM has access to data on all other PMs on its Universal Control Network. An AM has access to all data on the devices on an entire Local Control Network (LCN), including all Data Hiways and Universal Control Networks connected to the LCN.

3. The MC or PM has its own directly connected process I/O. The AM accesses process I/O through Hiway Gateways from devices on Data Hiways (see Figure 2-1). All data paths are redundant.

4. The MC and PM have special capabilities that make them well suited for discontinuous and batch applications, as well as for continuous-control applications. The version of CL in the AM is primarily intended for applications related to continuous control.

The MC or PM, therefore, is well-suited for smaller and more localized applications, and applications that involve batch or other types of discontinuous processes. The AM is well-suited for larger programs, larger amounts of data, and applications that require a wider scope of data access.

Although the differences between AMs and MCs and PMs were emphasized above to indicate when one or the other would be used, note that the functions that are common to AMs, MCs, and PMs appear the same to the user and to the rest of the system; thus, the engineer builds the database, configures standard functions, and implements programs for AMs, MCs, and PMs (and also builds the database and configures standard functions in all the other process-connected devices in TDC 3000X) in the same way, using the same facilities at a Universal Station. Any data from any device can be intermixed in any operating displays, and the source of information that an operator is using is normally of no concern to the operator. Control strategies that have an upper layer of their hierarchy in an AM and a lower layer in an MC or PM (or other process-connected device) are easily and seamlessly built and operated.

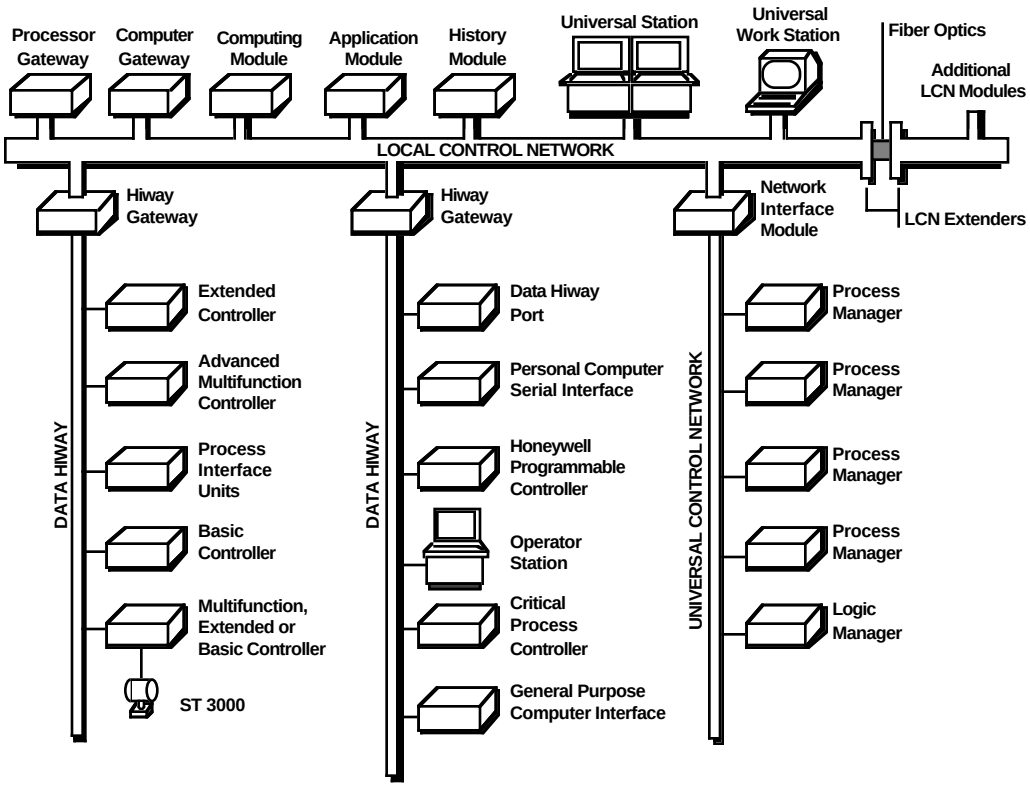


Figure 2-1 — Example TDC 3000^X System 1849

CL ON APPLICATION MODULES
Section 3

This section tells you how to use CL/AM to implement custom control strategies in TDC 3000X Systems.

3.1 CL/AM USE

A great deal of control functionality can be implemented, using standard capabilities in the AM without even using CL. In fact, most of the application work on most systems is typically accomplished by simply configuring the built-in capabilities. CL/AM is important, not so much because it is often needed but because CL/AM makes the system open-ended, which allows you to accomplish almost any kind of unique or customized control functionality in the relatively few but important cases where the standard capabilities don't do the whole job.

CL/AM can take over from the standard capabilities as much or as little of the job as is necessary. For example, if a CL/AM program is written to calculate a rate of reaction every five seconds so that it can be displayed for the operator or used in a control scheme, there are still many standard services provided by the AM that can be used. These include data access by name from anywhere in the system, initialization facilities, alarm checking and reporting, access from displays, bad-value notification, ability to install and modify the calculation with the rest of the system on-line, various scheduling options, etc. As another example, if a CL/AM program is written to carry out a nonstandard control algorithm, the data can be included in standard control displays and the system can provide deviation and output alarms, antiwindup propagation, back-initialization, etc. If these standard features were not provided, such as is the case with a general-purpose computer connected into a control system, the time required to provide the necessary substitutes would, in many cases, be much greater than the time spent on the actual application.

With this motivation in mind, before getting into CL/AM itself, we will briefly look at the standard functions that are available (complete information can be found in the *Application Module Control Functions* and the *Application Module Algorithm Engineering Data*). To do this, we will first examine the structure of the database, which is important as a background for both the standard functions and CL/AM.

3.2 DATABASE

The main structure in the AM database (as well as the database for all the TDC 3000X data acquisition and control devices) is a record of data called a data point. Generally, a data point is a collection of all the data involved with the acquisition or calculation of a main item of information and may optionally include the data for an associated control loop. In a data point in an AM, the main item of information is often a calculated quantity, such as a heat-flow rate, material-balance closure, motor-duty cycle, reaction rate, etc. Data points in process-connected devices are usually much simpler and the main item of information is usually an analog or digital measurement or a simple combination of these measurements.

Each data point has a name and a description. The main item of information for most types of data points is called the PV, which stands for process value or point value. There are usually a number of other associated items of data, such as the units of measure, high-alarm and low-alarm limits, pointers to input and output data, scheduling interval, etc. Some examples of the kinds of data contained in data points that do not implement a control loop are shown in Figure 3-1.

A data point may also contain additional data to form a control loop. In this case, the PV is the variable to be controlled, and the data point also typically includes a setpoint, control output, control mode, etc. Figure 3-2 shows an example of a control data point.

As shown in Figures 3-1 and 3-2, each of the data items in a data point is called a parameter of the data point and has a parameter name, such as PV, SP, PVLOTP, MODE, etc. Every data point in a system has a unique name but the same parameter name often appears in many data points; thus, both a data point name and parameter name must be given to uniquely specify a particular item of data. The designation of an item of data looks like

PTNAME.PARNAME

where PTNAME is the data point name and PARNAME is the parameter name, as shown in the following examples:

Name	Possible Value
FC105.PV	52.76
TOPTEMP.SP	610.00
TOPTEMP.PV	609.97
RXRATE.PVLOTP	20.50
UXCHNGR1.PTDESC	"XCHNGR NO. 1 HEAT XFER COEF."
FC105.MODE	AUTO

Data points are built at a Universal Station in the Engineering Personality, using the Data Entity Builder. Instructions are given in the *Data Entity Builder Manual*, in the *Implementation/Engineering Operations - 1* binder, which also provides references to other manuals that can be consulted to determine what information is contained in the various types of data points.

3.3 STANDARD CAPABILITIES

The built-in capabilities of the AM are invoked simply by including appropriate parameters in a data point and assigning appropriate values when a data point is built. For example, to invoke alarming of the PV if it goes above 250, the parameter PVHITP (PV high alarm trip value) is set to 250 and the parameter ALENBST (alarm enable status) is set to ENABLE. The parameter-entry displays of the Data Entity Builder contain descriptions of all the parameter entries.

There are two parameters of special interest because they can be used to select calculational and control functions from a library of built-in algorithms:

Parameter	Description
PVALGID	Name of algorithm to calculate the PV
CTLALGID	Name of algorithm to calculate control output

The *Application Module Algorithm Engineering Data* manual, in the *Implementation/ Application Module - 1* binder, contains a list of the PV and Control algorithms that are available.

When you select a PV or control algorithm, the Data Entity Builder automatically appends to the data point the additional parameters that are needed by that algorithm. For example, if a PID control algorithm is specified (CTLALGID = PID), parameters for the setpoint, output, tuning constants, and others are appended to the data point.

Parameter	Description	Possible Value
NAME	Name of data point	BLRIEFF
PTDESC	Data Point description	BOILER 1 EFFICIENCY
EUDESC	Units of measure	%
PV	Data Point value	85.62
PVLOTP	Low alarm trip	82.00
PKGNAME	Package for calculation	HTLOSSEF
GISRC(1)	Source of input 1	TI1024.PV
GISRC(2)	Source of input 2	ANLYZR1.PV
PERIOD	Execution interval	2SEC
etc.		

Figure 3-1 — Data Point Example

Parameter	Description	Possible Value
NAME	Name of data point	RFXC3
PTDESC	Data Point description	DEBUTANIZER REFLUX
EUDESC	Units of measure	KLB/HR
PV	Data Point value	110.6
SP	Setpoint	110.0
OP	Output	52.34
MODE	Control mode	MAN or AUTO or CAS or NORMAL

Figure 3-2 — Control Data Point Example

The value assigned to the parameter PERIOD determines the interval at which the data point is processed. Data point processing consists of obtaining inputs, calculating the PV by the specified PV algorithm, checking alarms, calculating the control output if a control algorithm is specified, and various other support actions. See the *Application Module Control Functions* for more information on data point processing in the AM.

3.4 CL/AM EXECUTION

If the standard data point-processing functions or the capabilities of the standard PV and control algorithms are not sufficient for a given requirement, CL/AM programs can be used to extend the built-in capabilities. There are three ways this can be done:

1. If more calculations and logic are needed than can be supplied by a standard PV algorithm, a CL/AM program can be used in place of a built-in PV algorithm by setting PVALGID = CL and linking the CL/AM program to the data point.

2. If control actions not supplied by a standard control algorithm are needed, a CL/AM program can be used in place of a built-in control algorithm by setting CTLALGID = CL and linking the CL/AM program to the data point.

3. CL/AM programs can also be linked to a data point (of type Regulatory Control, Custom, or Switch) to augment the other data point-processing functions, such as taking special action if an alarm limit is reached. Table 3-1 shows the different points in the processing sequence where CL/AM programs can be inserted. For unusually complex situations, more than one CL/AM program can be linked at any insertion point.

Any active CL/AM programs attached to a data point are executed when the data point is processed. The execution can be made conditional.

CL programs inserted at a data point's BACKGRND insertion point are referred to as Background CL programs; programs inserted at any other insertion point are referred to as Foreground CL programs. Foreground CL programs run as subroutines, and thus should execute within a single processing cycle for the data point. Background CL programs run in AM "free" time and can run over many processing cycles.

3.5 CL/AM PROGRAM EXAMPLE

To introduce the concepts that are needed to write CL/AM programs, we will follow a simple example from functional requirement to a working program. Assume that you want to calculate the ratio of internal vapor to liquid-flow rates on tray 5 in the fractionation column shown in Figure 3-3. The calculation is to be repeated every 5 seconds to update the value of the vapor/liquid ratio (this value might be used in a control scheme or just for operator information). Measurements are available for the top temperature, reflux temperature, tray-5 temperature, reflux-flow rate and product-draw-flow rate.

Table 3-1 — Insertion Points for CL/AM Blocks

Point Type	Insertion Slot	Description
Regulatory Control	PRE_GI	Before General Inputs are input.
	PRE_PVPR	Before any PV processing.
	PRE_PVAG	Before PV algorithm is executed.
	PV_ALG	Instead of standard PV algorithm (specify CL for PV algorithm).
	PST_PVAG	After PV algorithm is executed.
	PST_PVFL	After PV filtering.
	PRE_PVA	Before PV alarm checking.
	PST_PVPR	After all PV processing.
	PRE_CTPR	Before any control processing.
	PRE_SP	Before SP (target) and deviation alarm checking.
	PRE_CTAG	Before control algorithm is executed.
	CTL_ALG	Instead of standard control algorithm (specify CL for control algorithm).
	PST_CTAG	After control algorithm is executed.
	PST_CTPR	After all control processing.
	PST_GO	After General Outputs are output.
	BACKGRND	Background CL insertion point.
Custom	GENERAL	Foreground CL insertion point.
	BACKGRND	Background CL insertion point.
Switch	GENERAL	Foreground CL insertion point.
	BACKGRND	Background CL insertion point.

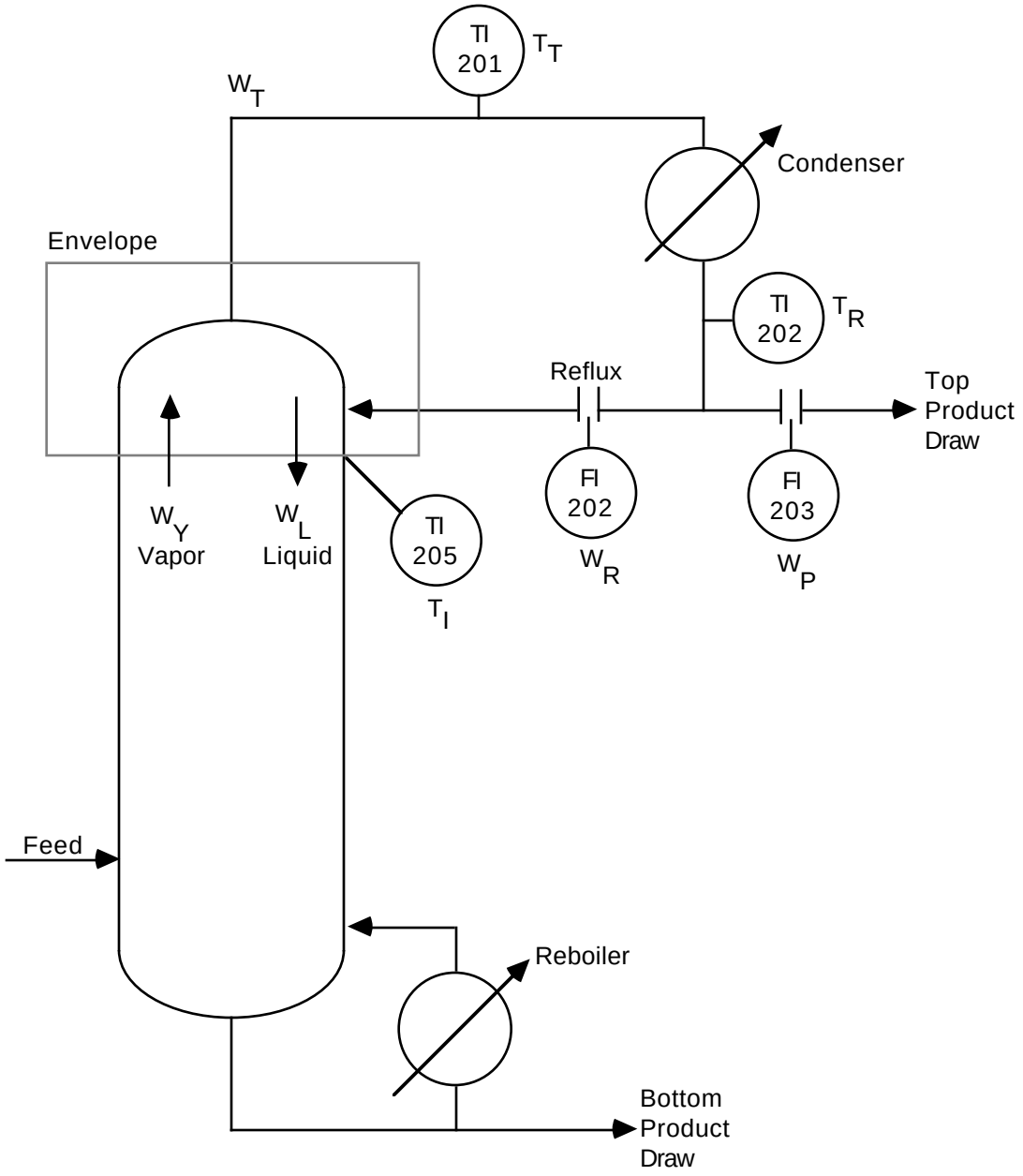


Figure 3-3 — Fractionation Column

1373

The material in the top of the column is such that the heat capacity of liquid and the heat of vaporization do not significantly change from tray 5 to the top of the column. By doing a simultaneous heat and material balance around the envelope indicated by dashed lines in Figure 3-3, we derive the following relationship for the liquid-flow rate on tray 5:

$$w_L = [Cp/lambda\ (T_T - T_R) + 1]w_R + Cp/lambda(T_T - T_I)w_p$$

where:

w_L	= weight-flow rate of liquid through tray 5
w_R	= external reflux-flow rate
w_P	= product-draw-flow rate
C_P	= heat capacity of liquid
lambda	= heat of vaporization
T_T	= top vapor temperature
T_R	= external reflux temperature
T_I	= temperature on tray 5

A material balance yields the vapor-flow rate through tray 5:

$$w_V = w_P + w_L$$

and the vapor/liquid ratio is:

$$R = w_V/w_L$$

The measured quantities are input by devices on a Data Hiway. Data points for these quantities have already been built (refer to *Data Entity Builder Manual* for details on data point building) and are named as follows:

Data Point Name	Measurement
TI201	T_T - top temperature
TI202	T_R - reflux temperature
TI205	T_I - tray 5 temperature
FI202	w_R - reflux flow rate
FI203	w_P - product draw flow rate

3.5.1 Data Point for the Example Program

Next we will build a data point that we will call VLRATIO to serve as database and schedule mechanism for our calculation. The vapor/liquid ratio R will be stored as the PV of this data point. The data point will reside in the AM and will be a regulatory type of data point. Regulatory data points are used when the PV is a continuous type of variable.

The Data Entity Builder (DEB) is used to build the data point. Point VLRATIO is built just like any AM Regulatory point that doesn't use a CL/AM program, except for these parameters:

PTDISCL = Full (disclosure for DEB displays)

CLSLOTS = 1 (number of CL/AM programs attached to the data point)

PVALGID = CL (PV algorithm name)

CLSLOTS indicates that one CL/AM program is to be bound to this point and PVALGID indicates that the CL/AM program, rather than one of the built-in algorithms, is to do the PV processing. PTDISCL must equal Full in order for CLSLOTS to appear in the PED (Parameter Entry Display).

The DEB is used to load VLRATIO into the AM. If the system has more than one AM, the AM into which VLRATIO is loaded is determined by the process unit. In this example, the process unit is C2, so parameter UNIT contains C2. All of the process units in the system are defined as the system is configured. Unit names are configured in Task 7 in the *System Startup Guide*, in the *Implementation/Startup & Reconfiguration - 1* binder.

When the DEB loads VLRATIO into the AM, it is left in an inactive state (PTEXECST = Inactive) and, at this time, it is incomplete because the CL/AM program that will calculate its PV is not yet available. The next steps, therefore, are to enter, compile, and link the program that will calculate the vapor/liquid ratio.

3.5.2 Program Statements

The complete program is shown in Figure 3-4. A CL/AM program is called a CL block. A block header, which gives the name that you wish to assign to the block and information about how it is to be executed, is the first line of a block. In our example, the block header is:

```
BLOCK vlr_calc (POINT vlratio; AT PV_ALG)
```

We decided to call the block VLR_CALC, and this name is entered after BLOCK. The identifier that follows POINT is the name of the data point that executes this CL block, which in our case is VLRATIO. The identifier that follows AT tells where we want the CL block to be executed in the point-processing sequence. In this case, PV_ALG indicates that the block will be executed in place of a built-in PV algorithm. See Table 3-1 for a complete list of the different insertion points for CL/AM programs.

The CL compiler makes no distinction between lower-case and upper-case letters; thus, for the data-point name we could write VLRATIO, vlratio, or VLratio. In programs in this manual, upper case is used for identifiers that are part of the CL language or are standard system names; lower case is used for identifiers that are chosen by the user. Identifiers can consist of letters, numbers, nonconsecutive underscores, and certain other special characters, and must include at least one non-numeric character (see *Control Language/Application Module Reference Manual* for a complete description of legal identifiers). Identifiers that reference data from outside the CL/AM program are restricted to a maximum of eight characters, but identifiers that are local to the program (see below) can be any length that will fit on one line.

You can put comments anywhere in a CL program by preceding the comment with a double dash (--). Anything from the double dash to the end of a line is ignored by the CL compiler.

```
BLOCK vlr_calc (POINT vlratio; AT PV_ALG)
--
--   Calculate internal vapor/liquid ratio for column no. 2.
--
EXTERNAL ti201, ti202, ti205, fi202, fi203
LOCAL wl, wv, ratio
LOCAL cp = 0.806    -- liquid heat capacity
LOCAL lambda = 653.0 -- heat of vaporization
--
--   Calculate internal liquid flow rate from heat and material
     balance
SET wl = ((ti201.PV - ti202.PV)  *  cp/lambda + 1)  *  fi202.PV
&       + (ti201.PV - ti205.PV)  *  fi203.PV  *  cp/lambda
--
--   Calculate internal vapor flow rate from material balance
SET wv = fi203.PV + w1
--
--   Calculate vapor/liquid ratio and store as PV
SET ratio = wv/w1
--
CALL ALLOW_BAD (PVCALC, ratio)
SET PVCALC = ratio
--
END vlr_calc
```

Figure 3-4 — CL/AM Program VLR_CALC

Following the header, you must declare the residing places of the data used by the block:

EXTERNAL – The data is contained in a different data point than the data point to which this block is attached.

PARAMETER – The data is contained in a parameter of the data point to which this block is attached. This statement is required only for generic programs or to declare parameters of the "point" that is not built yet.

LOCAL – The data exists only while the block is executing, has no visibility from other data points, programs, or operator displays, and is not saved from one execution to the next.

For our example, the declarations are

```
EXTERNAL ti201, ti202, ti205, fi202, fi203

LOCAL wl, wv, ratio
```

This says that we will access one or more parameters from data points TI201, TI202, TI205, FI202 and FI203, and we will have three local variables WL, WV and RATIO to hold temporary results.

There can be any number of declaration statements, each declaring one or more variables, and they can be in any order. For example, the first declaration above could have been written

```
EXTERNAL ti201, ti202
EXTERNAL ti205, fi203, fi205
```

Constants can be defined by a variation of the LOCAL statement:

```
LOCAL cp = 0.806
LOCAL lambda = 653.0
```

We also use the parameters PVCALC and PVAUTOST, which are on the block's own data point (VLRATIO). It is not necessary to declare these parameters because the compiler can determine that they exist on the data point that we named in the block header.

Next come the statements that define the calculations and logic to be carried out when the block is executed. For our example, the statements are simply the arithmetic calculations that we derived earlier:

```
SET wl = ((ti201.PV - ti202.PV)  *  cp/lambda + 1)  *  fi202.PV
&  + (ti201.PV - ti205.PV)  *  fi203.PV  *  cp/lambda

SET wv = fi203.PV + wl

SET ratio = wv/wl
```

These are called assignment statements; they evaluate the expression on the right of the equal sign and store the resultant value in the variable on the left. The statements are evaluated in the order that they appear.

Note that & in column 1 is used to indicate a line that is a continuation of the previous line; otherwise, each line starts a new statement. Other than the &, all CL statements are free-format. Spaces and indentation can be used as desired for readability, except that at least one space must appear between adjacent identifiers, and spaces and continuation-line breaks cannot appear within identifiers or numbers.

As mentioned earlier, parameters on points are accessed by writing the point name followed by a dot followed by the parameter name; thus, TI201.PV refers to the value of the PV of TI201, which in this case is a measured temperature. Values can be assigned to parameters on other points by putting the point and parameter name on the left of the equal sign:

```
SET ti201.SP = (ti204.PVHITP + ti204.PVLOTP)/2 + offset
```

The data point to which a CL block is attached is referred to as the "bound" data point, because the process of linking the CL block to the data point (described later) can be thought of as binding the CL block and the data point together. When parameters of the bound data point are accessed in a CL/AM program, the parameter is not preceded by the data-point name and dot, as is done with parameters of external points. In our example, PVCALC and PVAUTOST are parameters of the bound data point (VLRATIO), so PVCALC and PVAUTOST are used by themselves rather than as VLRATIO.PVCALC and VLRATIO.PVAUTOST. (The use of these parameters is explained later.)

In our example program in Figure 3-4, the local constants CP and LAMBDA are not really required; the constants 0.806 and 653.0 (or their quotient) could be written in the assignment statements in place of CP and LAMBDA. It is, however, good practice to name the constants, as is done in this example, to facilitate later program updates that may be required, particularly if a constant is used in more than one place in a program.

The intermediate variables WL and WV are not required. The three assignment statements could be algebraically combined into one expression for the vapor/liquid ratio that is stored into RATIO; however, the use of the intermediate variables makes the calculations a little clearer. In longer calculations where a complex subexpression appears in several places, assignment of the subexpression to an intermediate variable before the subexpression is used can keep a program from getting hopelessly confusing, and save processing time because the subexpression is evaluated only once.

We want RATIO to be the PV of the data point VLRATIO. To do this, the program must store RATIO into the standard parameter PVCALC rather than PV. This is because the system moves the value from PVCALC to PV during data-point processing if the PVSOURCE of the data point is set to AUTO (automatic). If there is a problem with input data, for example, the operator can manually change PVSOURCE to MAN (manual) and then enter an approximate value of the vapor/liquid ratio. The operator-entered value goes to a parameter called PVMAN and the system moves PVMAN to PV if PVSOURCE is MAN. PVSOURCE can be thought of as a switch that directs either PVCALC or PVMAN to PV.

If data that is input by the program has not been defined or is inaccessible (e.g., because of a sensor failure) when the program is executed, the system sets the data value to a special code that signifies a bad value. Bad values propagate through calculations. In Figure 3-4, this means that RATIO would have a bad value if any of the PVs of TI201, TI202, TI205, FI202, or FI203 is bad.

If a program attempts to store a bad value to a parameter of its own data point or an external data point (or if a program uses a bad value in a comparison, which will be discussed later), the program is aborted. An abort means that the store does not take place, an exit from the program is forced, and an alarm condition is raised. The program continues to be executed at its scheduled interval and if the bad value becomes okay (e.g., because a sensor is fixed) the program executes normally and the alarm condition is dropped.

In the example in Figure 3-4, the subroutine ALLOW BAD is used to ensure that the program is not aborted. The statement

```
CALL ALLOW BAD (PVCALC, ratio)
```

stores the value of the subroutine's second argument (in this case ratio) into the subroutine's first argument (in this case PVCALC). In this respect, the subroutine call could be replaced by

```
SET PVCALC = ratio
```

If the value of ratio is bad, however, the SET statement aborts the program.

The subroutine ALLOW BAD moves the value from ratio to PVCALC whether or not the value is bad and does not abort the program.

The final statement of the CL block is

```
END vlr_calc
```

which includes the name we assigned to the CL block in the BLOCK statement. This completes the program. It also serves as the execution exit when, as in this program, there is no explicit EXIT statement.

At this point the program exists only on paper. To get it into the system and working, we need to do the following steps:

1. Enter the source statements, which are shown in Figure 3-4, into a file on the system.
2. Compile the program source statements to produce an "object" file that can be directly executed by the AM.
3. Load the program object file into the AM by linking it to the data point called VLRATIO that we previously built.
4. Change the execution mode of the VLRATIO data point from INACTIVE to ACTIVE.

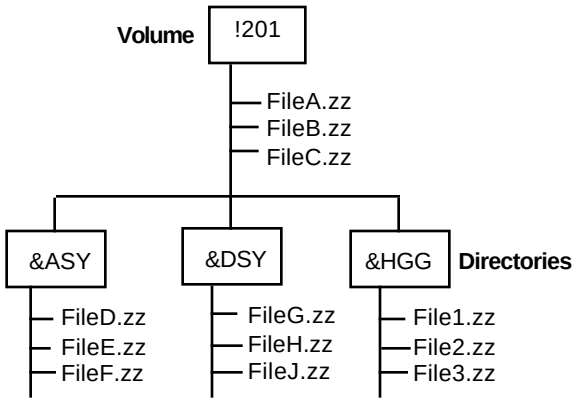
These steps are explained under 3.6, 3.7, 3.8, and 3.9.

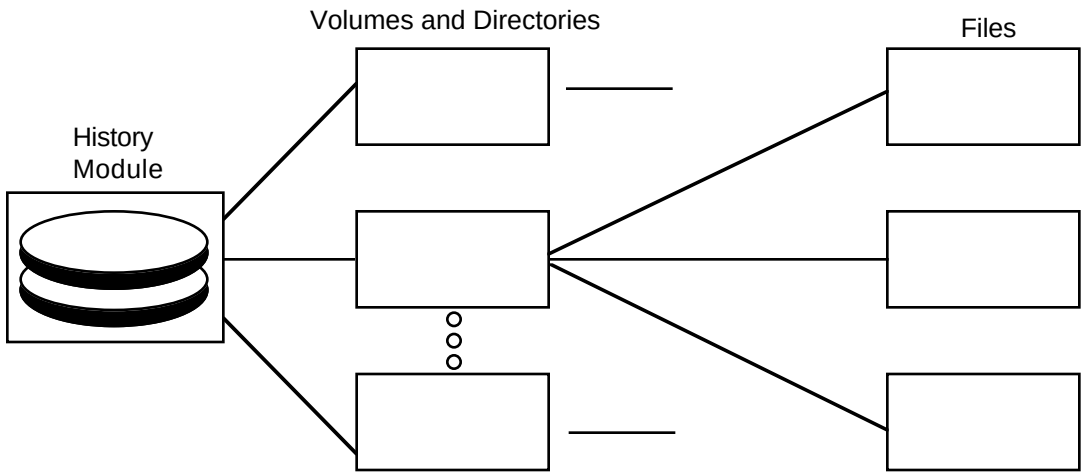
3.6 FILE CONCEPTS

We first need to review the concept of a file. Files on TDC 3000^X are used to hold many different kinds of things, such as display definitions, data-point definitions, area definitions, etc. At present we want to use a file to hold the source statements for a CL/AM program.

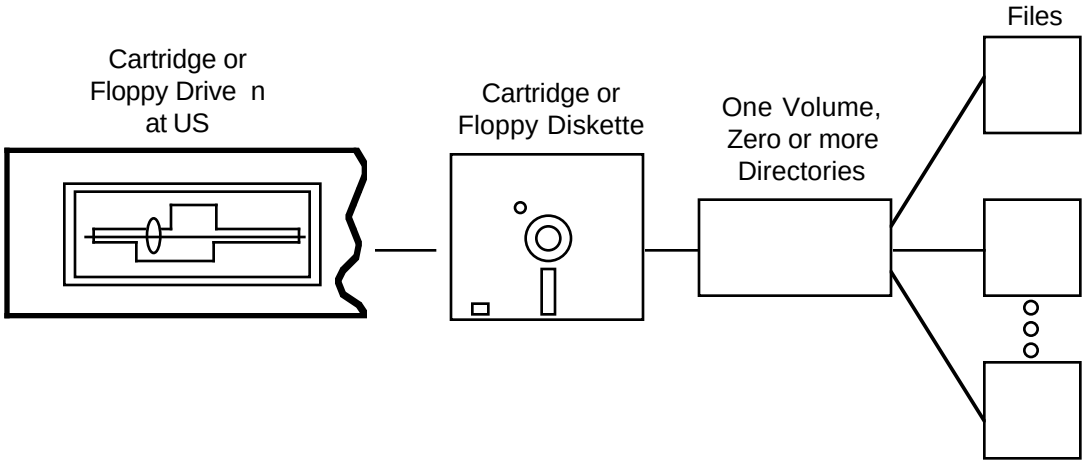
Files are contained in volumes and directories, and the files, volumes, and directories are stored on History Modules (HMs) or on removable media. Two types of removable media are available—cartridges and floppy disks.

This sketch shows how files, volumes, and directories relate to each other. A volume can contain one or more files and one or more directories. Each cartridge and each floppy disk is a volume and has a volume name. HMs can have more than one volume.





Net>Vldr>Filename



$Fn>Vldr>Filename

Vldr = Volume or directory name.

Figure 3-5 — File, Volume, and Device Organization 1374

For CL program files, we need a volume that has been set aside for user files. This user volume can be on either an HM or a removable medium. The File Utilities program is used to create a volume on a diskette, as explained in Task 4 in the *System Startup Guide*. If the diskette has never been formatted on a TDC 3000^X System it must be formatted before it can be used; this is also explained in Task 4 in the *System Startup Guide*.

Creating a volume on an HM is somewhat more involved. One or more user volumes should be created for future needs when the system is first configured; see the *Network Form Instructions* in the *Implementation/Startup & Reconfiguration - 1* binder.

Files are identified by a name chosen by the user and an extension that usually describes the type of contents of the file. The extension is usually dictated by the use of the file. The complete designation of a particular file and where it is located is called a pathname, and looks like

```
device>vol>filename.ext
```

The pathname parts are

ext —	the extension to the filename
filename—	the file name chosen by the user
vol —	the volume name
device—	NET if the file is on an HM, or the physical device identification if the file is on a cartridge or floppy disk.

For the file for our example program, we assume that a volume called USR1 on an HM is used. For simplicity we name the file the same as the block, VLR_CALC, but any file name could be used. The CL compiler requires that the extension of any CL source file to be compiled must be "CL"; therefore, the complete pathname of our source file is

```
NET>USR1>VLR_CALC.CL
```

3.6.1 Program Source Entry

The Text Editor is used to create the source file for the program. It is also used to correct or alter the source file. Instructions for creating CL/AM source files are in the *Control Language/Application Module Data Entry* publication, in the *Implementation/Application Module - 2* binder. Briefly, operation is as follows:

1. At a US in the Engineering Personality mode, select the COMMAND PROCESSOR pick from the Main Menu.

2. Key in ED followed by the pathname (for our example: `ED NET>USR1>VLR_CALC.CL`). Press ENTER.

3. An error message appears indicating that the file is presently empty (which it should be). Just ignore this; it goes away as soon as you type something.

4. Type the source statements, just as they appear in Figure 3-4 for our example. End each line with a <CR> (carriage return key).

5. To correct mistakes, the <DEL> key (not <DEL CHAR>) deletes the character under the cursor, and <LF> (linefeed key) followed by I toggles between insert and overwrite modes. The cursor is moved by holding <CTL> down and pressing the arrow keys. To page forward, type <LF> and then press <CTL>; to page backward, type <LF> and then press <CTL>.

6. Press HELP to access a help display that describes all the editor features. This display is paged by using the up and down arrows by themselves. To get back to your editing, press CANCEL.

7. To stop working on the file at any time, type <LF> E <CR>. The entries and corrections you have made are saved in the file whose pathname you entered in Step 2. To exit the editor, hold CTL and press the MENU key. To enter more material or make corrections at a later time, repeat Steps 1 and 2 to call up the previously saved material and begin editing.

It is a good idea to save a backup copy of a file on a floppy diskette whenever you enter enough material, so that you would not have to reenter it if some problem were to occur that destroyed the file. The File Utilities program discussed earlier is used to copy files. An example command is

```
COPY NET>USR1>VLR_CALC.CL  $F1>BKUP>VLR_CALC
```

This copies VLR_CALC.CL from the HM to a volume called BKUP on the diskette in drive 1. Note that no extension is entered for the destination file; the destination file is automatically assigned the same extension that the source file has.

File Utilities can also be used to display or print files. For our example file, the command is

```
PRINT NET>USR1>VLR_CALC.CL
```

The destination for output from this command is determined by the last DATAOUT command that is entered:

```
DATAOUT        - directs the output to the screen
DATAOUT $Pn    - directs the output to printer n
```

The PAGE keys can be used to page through file output that has been directed to the screen.

3.7 COMPILATION

After the source for the CL/AM program has been entered, the CL compiler is used to produce object code that can be loaded into and executed by the AM.

The CL compiler expects to find the program source file on a preselected device and volume. You can choose this device and volume by selecting the Utilities target from the Main Menu of the Engineering Personality and selecting the Modify Volume Paths target from the next menu. This results in a display of all the preselected devices and volumes for various TDC 3000^X functions. Touch or tab to the window labeled CL Source/Obj and enter the device and volume, which for our example is NET>USR1>.

You invoke the CL compiler from the Main Menu of the Engineering Personality by selecting the COMMAND PROCESSOR pick, then key in the following command (and press ENTER) to compile our example program:

```
CL VLR_CALC
```

where VLR_CALC is the name of the file that contains the source program. For our example, the file name and the CL block name are the same, but if they were different the file name, not the block name, must be entered.

The compiler takes the device and volume from the last entry made on the Modify Volume Paths display and the extension is assumed to be CL, so enter only the file name itself.

The compiler outputs messages indicating the phases of the compilation that are taking place and ends with a message showing either the number of errors found during the compilation, or that there were no errors. If there were errors (there always are the first few times a new program of any size is compiled), the compiler generates an error-listing file that contains the source statements from the NET>USR1>VLR_CALC.CL file and error messages after each statement found to contain an error. This error-listing file has the same name as the program source file except that the extension is .LE instead of .CL.

To view the errors, enter the command

```
PRINT NET>USR1>VLR_CALC.LE
```

This displays the error-listing file on the screen. You can page back and forth using the PAGE keys. If the listing is long and contains a lot of errors, you might want a paper copy to work from. To obtain a listing, enter

```
DATAOUT $Pn
```

where n is your printer number, and then repeat the above PRINT command.

To correct the errors, you can call the Text Editor program directly from the compiler by typing `ED NET>USR1>VLR_CALC.CL.` After correcting the program, use the CL compiler as previously described to recompile the program (you do not have to reselect the default device and volume if they have not been altered). Repeat this procedure until there are no errors (help in correcting errors is available in Section 8 of the *Control Language/AM Data Entry* manual, in the *Implementation/AM - 2* binder.

When a compilation results in no errors, the compiler does not generate a .LE file; it generates a .LS file that is a listing of the original .CL file with line numbers and page breaks added and a cross reference that lists each variable name and shows where it was used in the program.

3.8 LINKING

When there are no compilation errors, you can link the object form of the program to the data point that you previously built and loaded into the AM (the data point name for our example is VLRATIO). You enter link commands from the Compiler/Linker options display (to get there from the Engineering Main Menu, select the COMMAND PROCESSOR pick, then key in CL and press ENTER). The command for our example is

```
LK VLR_CALC  VLRATIO
```

where VLR_CALC is the file name of the CL/AM program, just as for a compilation, and VLRATIO is the name of the point to which the program will be linked. All link errors are shown directly on the screen. Consult the *Control Language/AM Data Entry* manual to find the cause of the errors and make corrections.

3.9 ACTIVATING THE DATA POINT

When there are no link errors, go to the Operator Personality and set the data point VLRATIO to the ACTIVE state (one way this can be done is from the PTEXECST parameter of VLRATIO's Detail Display) and process special the point. The program now executes and stores the internal vapor/liquid ratio into the PV of VLRATIO at the frequency specified when the VLRATIO data point was built.

The PV can be built into and viewed from group displays and schematic displays. It is possible to add a control algorithm to the data point VLRATIO to control the vapor/liquid ratio at a specified setpoint. The ratio could also be input to other programs that do further column calculations. The system checks and displays alarms if the ratio exceeds limits and the value of the ratio can be historized.

3.10 SUMMARY OF CL/AM PROGRAM INSTALLATION

The procedure for adding a CL program in an AM is as follows:

1. Use the Data Entity Builder to build the data points that are accessed by the CL/AM Program (i.e., data points that appear in EXTERNAL statements in the program) if these data points do not exist.

2. Use the Data Entity Builder to build the data point to which the CL/AM Program will be attached if this data point does not exist.

3. Use the Text Editor to enter and correct the source statements for the CL/AM program in a file with extension .CL.

4. If the default device and volume used by the CL Programs are not set to the device and volume you are using for the source program file, use Modify Volume Paths to correctly set the default.

5. Use the Command Processor to compile the program, using the name of the file that contains the source program.

6. If there are compilation errors, view the file with extension .LE.

7. Repeat Steps 3-6 until there are no compilation errors.
8. Use the Command Processor to link the program to the data point.
9. From the Operators Personality, set the point ACTIVE and process special the point.

If desired, compilation and linking can be combined in one command:

```
CLK filename pointname
```

If there are compilation errors, the link does not take place.

The procedure for modifying an existing program is as follows:

1. Use Text Editor, the Command Processor, and File Manager Utilities to add to or modify the CL/AM program, recompile it, view the error listing, and correct errors, until there are no remaining compilation errors.
2. From the Operator Personality, set the data point to which the program is attached to the INACTIVE state.
3. Use the Command Processor to unlink the old CL block from the data point. The command is

   ```
   UNLK blockname pointname
   ```

   Note that the CL block name is used in the unlink command, while the CL program source-file name is used in the compile and link commands.
4. Use the the Command Processor to link the new CL block to the data point. The command is

   ```
   LK filename pointname
   ```

   See Note in Step 3.
5. From the Operator's Personality, set the data point ACTIVE and process special the point.

If you forget the commands used by the Command Processor, enter CL and press ENTER. The Command Processor then displays the CL compiler/linker commands and options.

3.11 CUSTOM DATA SEGMENTS

Looking back at the CL/AM program in Figure 3-4, the values of the heat capacity CP and the heat of vaporization LAMBDA are defined in a LOCAL statement in the program. The only way to see these values is to use File Manager Utilities or Text Editor to look at the program source file, and the only way to change these values is to edit them, then recompile and relink the CL/AM program. If these values need to be changed occasionally (for example, when the feedstock to the column changes), it would be nice to have the values more accessible. This can be done by building a Custom Data Segment, which allows you to define new parameters and add them to data points. Once you define new parameters and add them to a data point, they can be accessed just like a standard parameter, such as PV, SP, MODE, etc., from a Detail Display or a Schematic Display. The value of the new parameters can therefore be easily viewed and changed.

```
CUSTOM (CLASS PV_ALG)
--
--    Define parameters for vapor/liquid ratio calculation
--
PARAMETER cp "heat capacity"
      VALUE   0.75
      EU      "Btu/lb'F"
--
PARAMETER lambda "heat of vaporization"
      VALUE   650.0
      EU      "Btu/lb"
--
END CUSTOM
```

Figure 3-6 — Custom Data Segment for VLR_CALC

For our example, a Custom Data Segment (CDS) that defines two new parameters called CP and LAMBDA is shown in Figure 3-6. The first statement is the header for the CDS:

```
CUSTOM (CLASS PV_ALG)
```

The header does not include a name for a CDS because the system uses the name of the file in which the CDS source is compiled as the name of the CDS; this works because only one CDS can be contained in a file. The optional attribute (CLASS PV_ALG) determines where parameters of the CDS are displayed; the following are the possible choices:

PV_ALG, CTL_ALG—The parameters in the CDS are displayed on the PV Algorithm or Control Algorithm page, respectively, of the Detail Display for the data point to which the CDS is attached. (See heading 4.4.3.2 in the *Control Language/AM Reference Manual* for parameter display restrictions on the PV and Control Algorithm pages.)

GENERAL—The CDS parameters are displayed on the Custom Data Segment page of the Detail Display. This is the default condition when no class attribute is specified.

Each parameter that is being defined has a PARAMETER statement that defines the name of the parameter and its data type. For our example, these are

```
PARAMETER cp     "heat capacity"
PARAMETER lambda  "heat of vaporization"
```

where CP and LAMBDA are the parameter names.

If the data type is not named, a default of NUMBER is used.

The description in quotes serves a special purpose; later on, when a data point is built to which the parameters in this CDS are attached, the parameter name and the description appear on the Parameter Entry Display (PED) that is displayed by the Data Entity Builder. It may happen that the person who builds the data point that uses the parameters in the CDS may not be the same person who designed and built the CDS, so the description provides communication from the person who built the CDS to the person who is building the data point.

Optional statements that define attributes of the parameter follow each parameter statement. In our example, the parameter CP and its attributes are

```
PARAMETER cp          "heat capacity"

   VALUE               0.75
   EU                 "Btu/lb'F"
```

The VALUE attribute defines a default (starting) value. If this attribute is not used, the default value is BAD for data-type number. The value can be overwritten when the data point is built, by using the Data Entity Builder. Also, the value can be changed from displays in the Operator Personality after the data point is built and loaded.

The EU attribute defines engineering units of measure that appear on the Data Entity Builder's PED.

There are several other types of attributes; for a complete description see the *Control Language/AM Reference Manual.*

The CDS is terminated by END CUSTOM.

To get a CDS into the system so that its parameters can be included on data points, the source statements must be entered into a file and compiled, analogous to the procedure used for a CL program. The CDS in Figure 3-6 is entered into a file by using Text Editor, as described before for the CL/AM program in Figure 3-4. We will use the file name VLR_DATA. The complete pathname for our example is

```
NET>USR1>VLR_DATA.CL.
```

The CDS is compiled by using the Command Processor, as described before. Before using the compiler, use the Modify Volume Paths display to make sure that your user-volume is entered for CL Custom GDF and CL Source/Obj.

The compiler maintains a library of system files that contains the names of various things that the user has defined. One file in this library contains all nonstandard parameter names in every CDS that has ever been compiled. The file can hold 2000 names, which is normally more than adequate (particularly since any of these parameters can be arrays of values—arrays are discussed later, and a particular name is entered only once, no matter how many times it is used in multiple CDSs). Once a name is entered into the library file, however, there is no way to delete the name, and it is not desirable to clutter up the file with parameter names that were accidentally mistyped. The compiler, therefore, does not update the library file when a CDS is compiled unless the compiler directive -UL (Update Library) is invoked. The following is the sequence you should follow when compiling a CDS:

1. First compile without the -UL directive to make sure that the CDS parameters do not contain errors. The command for our example is

   ```
   CL VLR_DATA
   ```

   Every new parameter is followed by an error indicating that the -UL option should be used. If any other errors appear, they should be corrected.

2. Recompile with the -UL directive to update the system library file with the CDS parameter names. The command for our example is

   ```
   CL VLR_DATA -UL
   ```

   There should not be any errors.

3. It is a good idea on subsequent recompilations to compile without the -UL directive unless a new parameter name is purposely being added to the CDS. This guards against erroneous additions to the system library file that might occur if a parameter name in the CDS is accidentally changed while editing something else in the file.

You can see all the parameters that have ever been defined by using File Manager Utilities to print the system library file &ASY>PARAMETR.SP; the file &ASY>SEGMENTS.SP can be printed to see all the CDS names that have ever been used. The -UL directive also controls whether CDS names are entered into the library.

Compiling a CDS does not set aside storage for the parameter values; it simply defines the parameters to the system. The next step is to build a point that uses the CDS parameters. For our example, this is the VLRATIO data point.

The data point VLRATIO is built by using the Data Entity Builder as before, except that you set NOPKG (number of packages—this really means number of CDSs that are attached to the point) to 1. This causes the Data Entity Builder to display a window called PKGNAME(1). Enter the name of the package (or CDS) source file that was previously compiled, which for our example is

```
VLR_DATA.
```

The Data Entity Builder now displays the parameters that constitute the CDS, including the parameter name, default value if any, and description string if any. The only action you need to take is to enter new values if the default values are not correct. When the data point is successfully built and loaded into the AM, the CDS parameters are part of the data point's data, along with the standard parameters such as PV, PVHITP, etc.

The CL block that uses the newly defined parameters is shown in Figure 3-7. Comparing Figure 3-7 to the original program in Figure 3-4, note that the CL block no longer has LOCAL statements to assign storage and values for CP and LAMBDA because these variables are now parameters of the bound data point VLRATIO. It is not necessary to include CP and LAMBDA in declaration statements in the CL block because the compiler can locate these parameters on the data point VLRATIO. After the CDS in Figure 3-6 is compiled and the data point VLRATIO is built, enter and compile the CL block in Figure 3-7 just as previously described for the CL block in Figure 3-4. You can now link the CL block to the data point as previously described, set the point ACTIVE, and process special the point.

```
BLOCK vlr_calc (POINT vlratio; AT PV_ALG)
--
--    Calculate internal vapor/liquid ratio for column no. 2.
--
EXTERNAL ti201, ti202, ti205, fi202, fi203
LOCAL wl, wv, ratio
--
--    Calculate internal liquid flow rate from heat and material
      balance
SET wl = ((ti201.PV - ti202.PV)  *  cp/lambda + 1)  *  fi202.PV
&     + (ti201.PV - ti205.PV)  *  fi203.PV  *  cp/lambda
--
--    Calculate internal vapor flow rate from material balance
SET wv = fi203.PV + wl
--
--    Calculate vapor/liquid ratio and store as PV
SET ratio = wv/wl
--
CALL ALLOW_BAD (PVCALC, ratio)
--
END vlr_calc
```

Figure 3-7 — VLR_CALC Using CDS Parameters

If it is necessary later to add to or alter the CDS, recompile the CDS as previously described, rebuild and load the point by using the Data Entity Builder, recompile the CL block if it has been changed, set the point INACTIVE, unlink the CL blocks on the point, relink the CL blocks, set the point ACTIVE, and process special the point.

If another data point needs parameters called CP and LAMBDA with the same attributes as defined in the CDS in VLR_DATA, the CDS does not have to be recompiled. It is necessary only to enter the CDS filename when the other data point is built to attach the parameters to the other data point. After once compiling a CDS, its parameters can be included on any number of data points. Each parameter occurrence is a separate variable; thus, VLRATIO.CP and POINTX.CP are two different and unrelated variables that may have different values, just as TI201.PV and TI520.PV are two different and unrelated variables.

The CDS structure is a convenient way to group parameters together for compilation and attachment to data points. Once a data point is built, the parameters of the CDS are part of the data point and are undifferentiated from other parameters of the data point.

3.12 CL/AM PROGRAM EXAMPLES

3.12.1 Generic Package Example

Let us assume that we want to calculate the internal vapor/liquid ratio for six fractionation columns. The calculations done by our CL/AM program VLR_CALC can be used for all six columns, but the names of the temperature and flow-measurement data points that supply the input data will be different for each column. This means that we have to make six copies of the source file that contains the program (giving each copy a different file name), change the measurement data-point names in each copy, compile all six programs, and link all six programs to six data points that were previously built. We need the six data points to do alarm checking and possibly to control each calculated vapor/liquid ratio, but editing and compiling six essentially identical programs seems like an unnecessary burden.

Fortunately, it is possible to build one generic CL block (as opposed to the point-specific CL block that we built before) and link it to any number of similar data points. Figure 3-8 shows the CDS and CL block of our previous example generically redone.

In Figure 3-8, the CDS and CL block can each be entered into a separate file and separately compiled as we did for the example in Figures 3-6 and 3-7, but it is usually convenient to put the CDS together with the CL block that uses the CDS's parameters into a package that can be edited and compiled as one item. As shown in Figure 3-8, the only additional statements needed to form a package are the header PACKAGE and the final statement END PACKAGE.

```
PACKAGE
CUSTOM
--
--     Define parameters for vapor/liquid ratio calculation
--
PARAMETER cp                          "heat capacity"
VALUE  0.75
EU     "Btu/lb'F"
--
PARAMETER lambda                      "heat of vaporization"
VALUE  650.0
EU              "Btu/lb"
--
PARAMETER topt                        "top vapor temperature"
EU              "degF"
--
PARAMETER rfxt                        "external reflux return temperature"
EU              "degF"
--
PARAMETER trayt                       "temperature of tray where V/L ratio is
                                       calculated"
EU              "degF"
--
PARAMETER rfxflow                     "external reflux flow rate"
EU              "lb/hr"
--
PARAMETER drawflow                    "top product draw flow rate"
EU              "lb/hr"
--
END CUSTOM

BLOCK vlr_calc (GENERIC; AT PV_ALG)
--
--     Calculate internal vapor/liquid ratio for any column
--
PARAMETER PVCALC
PARAMETER PVAUTOST: PVVALST
LOCAL w1, wv, ratio
--
--     Calculate internal liquid flow rate from heat and material balance
SET w1= ((topt - rfxt)  *  cp/lambda + 1)  *  rfxflow
&      + (topt - trayt)  *  drawflow  *  cp/lambda
--
--     Calculate internal vapor flow rate from material balance
SET wv = drawflow + w1
--
--     Calculate vapor/liquid ratio and store as PV
SET ratio = wv/w1
--
CALL ALLOW_BAD (PVCALC, ratio)
       --
END vlr_calc
END PACKAGE
```

Figure 3-8 — Generic Package VLR_PACK

Note that the bound data-point name in the CL-block header is replaced by the word GENERIC because the CL block will be bound to more than one data point. Also note that we must now declare the standard parameters of the bound data point that are used (PVCALC and PVAUTOST) and their data types, because the compiler has no bound data point to check for the existence of these parameters.

The main change is that all of the external data point parameter names in the CL block are replaced with names of parameters that have been added to the CDS. These parameters are on the data point to which the CL block is attached, so there are no external references. To highlight the difference between our previous point-specific program in Figure 3-7 and the generic version in Figure 3-8, the first assignment statement in the point-specific version of the program is:

```
SET wl =  ((ti201.PV - ti202.PV)  *  cp/lambda + 1)  *  fi202.PV
&         + (ti201.PV - ti205.PV)  *  fi203.PV  *  cp/lambda
```

and for the generic version, this statement is:

```
SET wl =   ((topt - rfxt)  *  cp/lambda + 1)  *  rfxflow
&          + (topt - trayt)  *  drawflow  *  cp/lambda
```

The variables TOPT, RFXT, RFXFLOW, TRAYT, and DRAWFLOW have been added to the CDS and are, therefore, parameters of the data point to which the program is attached.

But how do the measured values get into these new parameters? This is easily done by defining a General Input for each parameter when the data point is built. A General Input allows the user to specify a source data point and a parameter of the source data point, and a destination parameter of the data point being built. Each time the data point with the General Inputs is processed (executed) in the AM, the system obtains the input values from the specified source parameters from other data points and puts these values into the specified destination parameters of the data point being processed. This is done just before the CL/AM program is executed, so that the program can access the input values from other data points by simply accessing the data point's own parameters.

To add General Inputs to a data point being built for one of the six fractionation columns, set the PED port NOGINPTS (number of General Inputs) to 5 because there are five values to obtain from external sources. If the data point is being built for the column of our previous example, the following General Inputs are defined by entries in the Data Entity Builder PED:

i	GISRC (i)	GIDSTN (i)
1	TI201.PV	TOPT
2	TI202.PV	RFXT
3	TI205.PV	TRAYT
4	FI202.PV	RFXFLOW
5	FI203.PV	DRAWFLOW

where GISRC is the source data point and parameter and GIDSTN is the destination parameter on the data point being built. When you build the other five data points for the other five fractionation columns, the only difference (other than the data-point names and descriptions) are the data-point names that you enter for the General Input sources; these specify the unique measurements for each fractionation column.

You need to compile the package containing the CDS and CL/AM program only once. You then specify the CDS when building each of the six data points and link the CL/AM program to each of the six data points.

Even though a CL/AM program is linked to many data points, only one physical copy of the CL/AM program is loaded into the AM. The CL linker loads the CL/AM program to the AM the first time it is linked to a data point, but if the same program is linked to additional data points the linker simply sets up additional links to the one copy that was previously loaded.

3.12.2 Average Tank Temperature Example

Another example is given here to introduce a number of other features of CL/AM. The functional requirements for this example are as follows:

> For each of a number of tanks, calculate the average temperature of the liquid in the tank. Each tank has a level measurement and a number of temperature sensors located at different heights in the tank. Only the temperatures that are from sensors covered by the liquid and that are not bad measurements are to be included in the average. An indication of which temperature measurements are included in the average is to be provided. There is a maximum of six temperature measurements for each tank but some tanks have fewer.

Figure 3-9 shows a generic package that provides these functions. The package includes a CDS and a generic CL block.

In the CDS in Figure 3-9, notice that the parameters TTEMPS, HEIGHTS, and TUSED are arrays of values, rather than single values as we have previously used. The definition for TTEMPS is

```
PARAMETER ttemps:  ARRAY (1..6)
```

The numbers in parentheses indicate the range of an index that specifies individual elements of the array; thus, the TTEMPS array contains six values, and the individual values are referred to as TTEMPS(1), TTEMPS(2), ...,TTEMPS(6).

General Inputs are used to obtain the temperatures from other points and store the values in the TTEMPS array. One General Input is needed for each temperature measurement. The arrays are sized to handle a maximum of six temperature measurements for each tank. If a tank has fewer temperature measurements, the last value or values of the TTEMPS array are not obtained from General Inputs and therefore remain BAD values (because no VALUE attribute is used to enter values).

A General Input is also used to obtain the tank level and put it in LEVEL.

When the data points for each tank are built, the heights of the temperature sensors are entered into the HEIGHTS array. If the tank has fewer than six temperature measurements, -1 is entered in the unused elements of the HEIGHTS array. Note that corresponding elements of the TTEMPS and HEIGHTS arrays refer to the same sensor (e.g., HEIGHTS (5) is the height of the sensor whose value is input to TTEMPS (5)).

Every variable in a data point or a CL program is classified as a certain type of variable, depending on what kind of values the variable may take on. The values of the variables we have used so far have been numbers and are therefore of type NUMBER. (CL makes no distinction between integer and real values; both are considered type NUMBER.) When a variable is declared in a LOCAL or PARAMETER statement, the default type is NUMBER if no type is specified, so we haven't had to explicitly specify the type, although we could have. Both of the following declarations are the same, because NUMBER is the default type:

```
PARAMETER n
PARAMETER n: NUMBER
```

```
PACKAGE
-- Calculate the average temperature of the tank contents.
-- Include only those sensors that are covered by liquid and are not bad.
-- Store the average temperature as the PV.
--
CUSTOM
--
PARAMETER n                              "Number of temperature sensors"
VALUE 6
--
PARAMETER ttemps:  ARRAY (1..6)          "Tank temperature input values"
--
PARAMETER heights:  ARRAY (1..6)         "Height of each temperature sensor"
VALUE (5, 10, 15, 20, 25, 30)
--
PARAMETER level                          "Tank level input"
--
PARAMETER tused:  LOGICAL ARRAY (1..6)  "ON if corresponding T was used"
--
END CUSTOM

BLOCK ttcalc  (GENERIC; AT PV_ALG)
--
LOCAL tsum, nok, i
PARAMETER PVCALC
PARAMETER PVAUTOST: PVVALST
--
SET tsum = 0  -- sum of covered and OK temperature sensors
SET nok  = 0  -- number of temperatures in sum
--
sumloop: LOOP FOR i IN 1..n
IF (level > heights(i)) AND NOT BADVAL (ttemps(i))
&                                        AND (heights(i) >= 0)
&                                        THEN (SET tsum = tsum + ttemps(i);
&                                        SET nok = nok + 1;
&                                        SET tused(i) = ON)
ELSE  SET tused(i) = OFF
REPEAT sumloop
--
IF nok > 0  THEN SET PVCALC = tsum/nok
ELSE CALL SET BAD (PVCALC)
--
SET PVAUTOST = NORMAL
END ttcalc
END PACKAGE
```

Figure 3-9 — Package for Average Tank Temperature

In the CDS in Figure 3-9 the variables in the TUSED array are of type LOGICAL, as indicated by the declaration

```
PARAMETER tused: LOGICAL ARRAY (1..6)
```

LOGICAL variables can have only the values OFF and ON (you can also think of these values as meaning FALSE and TRUE). In our example, if TUSED(3) is set to ON by the CL/AM program, this says that the temperature that is input by the third General Input was used in the average. If TUSED(3) is set to OFF, this temperature was not used in the average.

The temperatures that are to be included in the average are summed into TSUM in a loop. Generically, a loop looks like

```
label:   LOOP FOR variable IN start val..last val

       statements

       REPEAT label
```

When the LOOP statement is executed it initially sets VARIABLE equal to START_VAL (which can be any arithmetic expression). The statements down to REPEAT are then executed, VARIABLE is incremented by 1, and the statements are executed again. This continues until VARIABLE exceeds the value of LAST_VAL. A unique label must appear in a pair of LOOP and REPEAT statements. The label is used to determine the scope of the loop when there is more than one (possibly nested) loop in a program.

In our example, the loop is started by

```
sumloop:  LOOP FOR i IN 1..n
```

and includes the ensuing statements down to

```
REPEAT sumloop
```

This LOOP statement repeats the statements in the loop with i successively set to values from 1 to N. The index of the temperature sensor is i.

The IF statement following the LOOP statement checks whether the height of sensor i (HEIGHTS(i)) is below the liquid level, and the value from sensor i (TTEMPS(i)) is not bad, and a positive value for the sensor height is entered. If all this is true, the statements in parentheses following THEN are executed, otherwise the statement following ELSE is executed.

If only one statement follows THEN or ELSE, the statement is written as if it stood alone. If more than one statement is to be conditionally executed following THEN or ELSE, the statements are enclosed in parentheses and separated by semicolons. These statements are all considered to be part of the IF--THEN or the ELSE statement, which accounts for the & signs used in Figure 3-9 to designate continuation lines. Note that THEN is considered part of the IF statement but ELSE starts a new statement. There is also an ELSE IF--THEN statement that can appear after an IF--THEN statement; see the *Control Language/AM Reference Manual* for more details.

In Figure 3-9, the system-supplied function BADVAL is used. This function yields the value ON (or TRUE) if the argument (TTEMPS(i) in the example) is a bad value. The system-supplied subroutine SET_BAD is also used. In the example, SET_BAD is used to store a BAD value into its argument PVCALC, if none of the temperature measurements pass the criteria for inclusion in the average.

Refer to the *Control Language/AM Reference Manual* for descriptions of other useful functions and subroutines that are supplied as part of the system. Also see applications product documentation about optional software packages that include additional subroutines and functions packaged as CL "Runtime Extensions." The *Control Language/AM Reference Manual* also describes how you can write your own subroutines and (1-line) functions.

3.12.3 Emergency Cooling Example

This example illustrates a CL block that is not used in a PV-algorithm slot or a control-algorithm slot. It is executed after the PV algorithm and just before the standard PV alarm-checking. The functional requirements are as follows:

> If the temperature that is input to TI200CHK.PV (by a standard PV algorithm) exceeds its high limit, open the valve on the cooling water sprays (VALVE20). When the temperature returns below its high limit, keep the valve open for a specified additional length of time and then close the valve.

Figure 3-10 shows a package that provides these functions. The CL block is executed in the PRE PVA slot (pre-PV alarm checking). The WHEN clause in the block header means that the CL block is executed only if the PV of the bound data point is greater than PVHITP (PV-high alarm trip value) or if the logical variable COOLING is ON. The WHEN clause can also be written:

```
WHEN PV > PVHITP OR cooling = ON
```

This program introduces variables of type TIME. A variable of type TIME can be an increment of time or an absolute time internally expressed as seconds-since-midnight of 1 Jan 1979. Externally, a value of type time can be written as any combination of hours, minutes, and seconds, such as

```
1 HRS 5 MINS 20 SECS

4 MINS

3.5 MINS

1 HRS 30 SECS
```

Each of the above lines represents <u>one</u> value of type TIME.

The function DATE TIME returns the present real time expressed as seconds since 1 Jan 1979. (TIME variables have sufficient capacity that they will not overflow until the year 2047.) When the temperature returns within limits, the program stores present time plus the specified delay DEL TIME into STP TIME, which is the time when the valve should be closed. When present time becomes greater than STP TIME the valve is closed.

```
PACKAGE
-- If TI200CHK temperature goes above limit, open cooling
-- water valve.  Keep valve open for a specified length
-- of time after the temperature returns below limit,
-- then shut valve.
CUSTOM (CLASS GENERAL)
--
PARAMETER cooling:  LOGICAL
VALUE OFF
--
PARAMETER timer:  LOGICAL
VALUE OFF
--
PARAMETER stp_time:  TIME
--
PARAMETER del_time:  TIME  "Cooling time after return below limit"
--
END CUSTOM

BLOCK coolit (POINT ti200chk; AT PRE_PVA; WHEN PV > PVHITP OR cooling)
EXTERNAL valve20
--
IF NOT cooling
& THEN  (SET valve20.OP = ON;       -- temperature is above limit;
&       SET cooling = ON)
     ELSE IF PV > PVHITP            -- cooling is ON
&    THEN SET timer = OFF           -- reset timer if PV > PVHITP
        ELSE IF timer = OFF         -- cooling is ON and PV <= PVHITP
&    THEN (SET timer = ON;          -- start timer
&       SET stp_time = DATE_TIME + del_time)
     ELSE IF DATE_TIME > stp_time
&       THEN (SET valve20.OP = OFF; -- time-out
&            SET cooling = OFF;
&            SET timer = OFF)
--
END coolit

END PACKAGE
```

Figure 3-10 — Package for Emergency Cooling

The cooling-water valve is opened and closed by a digital-output data point called VALVE20. Storing ON to the OP (output) of VALVE20 closes a digital-output contact that opens the valve; storing OFF does the reverse.

Although this program performs some simple timing and sequencing functions in a reasonably straightforward manner, CL in an MC environment provides tools that make timing and sequencing functions much easier to implement. Also, a timer-type of data point in the AM could be used in conjunction with the program to make the program simpler. Timer data points are discussed in the *Application Module Control Functions* manual.

3.13 PARAMETER LISTS

It may happen that a particular group of parameters is accessed by a number of different generic CL Blocks. To avoid the need to repeat the same PARAMETER statements in each block, a list of the parameters can be defined once by compiling a Parameter List separately from any CL Block; the name of the Parameter List can then be used in each block, rather than the individual parameter statements.

A Parameter List looks like

```
PARAM_LIST listname
   PARAMETER statements
END listname
```

The PARAMETER statements are exactly the same as those used within a block.

The Parameter List is compiled by the the Command Processor the same as CL Blocks and Custom Data Segments. The Parameter List can be compiled by itself or as part of a package. You must use the Modify Volume Paths display to enter your user volume for CL Param List before compiling a Parameter List. You must use the -UL switch (see suggestions in heading 3.11) in order to get a clean compilation.

Compiling a Parameter List simply causes the system to remember the parameters in the list for later use. To use a Parameter List, you put the name of the list after GENERIC in the block heading of any CL Block that uses some or all of the parameters in the list. For example,

```
BLOCK blr_eff (GENERIC blr_pl; AT PV_ALG)
```

where BLR_PL is the name of a Parameter List that is either in the same package with the block or was previously compiled. The program BLR_EFF now has access to all the parameters in the BLR_PL Parameter List just as if these parameters had appeared in PARAMETER statements in the program.

There is a parameter list called $REG_CTL that is furnished with the system, already compiled. It contains the parameters of regulatory-control type of points that are usually used by CL blocks. In the example in Figure 3-8, we could use the following block header:

```
BLOCK vlr_calc (GENERIC $REG_CTL; AT PV_ALG)
```

and omit the PARAMETER statements in the block. See the *Control Language/AM Reference Manual* for a list of all the parameters in the $REG_CTL list.

3.14 DATA TYPES

We have so far encountered three types of data: number, logical, and time. In the next sections we will look at the remaining data types, which are data point identifier, enumeration, and string.

3.14.1 Type Data Point Identifier

A variable of type data point identifier contains the name of a data point. This type of variable is primarily used in generic CL Blocks to indirectly reference parameters from external data points.

To illustrate the use of variables of type data point identifier, Figure 3-11 shows the package of Figure 3-9 redone to use indirect reference rather than General Inputs to access the tank-temperature values. In the CDS, the TTEMPS array is redefined as

```
PARAMETER ttemps: $REG_CTL ARRAY (1..6)
```

Note that the name of a Parameter List, in this case $REG CTL, appears in the field where we have previously used the name of a data type (such as NUMBER, LOGICAL or TIME). The appearance of a Parameter List name in the data type field does two things:

1. It defines that the parameter is of type data point identifier, or in other words, the value of the parameter is a data point name.

2. It tells the system that any data point whose name is later stored into the parameter may have some or all of the parameters that are in the specified Parameter List (the data point can have additional parameters as well). This gives the compiler a way to determine the data types of parameters that are indirectly referenced.

A data point name is stored into a type data point identifier Custom Data Segment parameter when:

a) you build or rebuild the Bound Data Point and a CL package is attached with parameters that are of type data point identifier.

b) the Custom Data Segment data point identifier is changed through the Detail Display's Custom page or through a schematic actor.

c) the CDS data point identifier is changed through a Computer Gateway (CG) or AM General Input/Output (GI/GO).

d) the CDS data point identifier is changed through CL MOVE_PARAMETER and SET_NULL_POINT_ID subroutines.

In the CL Block in Figure 3-11, the action taken when TTEMPS(I).PV is referenced is as follows:

1. Take the case where I = 1. The data point name in TTEMPS(1) is accessed. This is the name that was stored in TTEMPS(1) when the bound data point was built. Assume that this name is TI1105.

2. The value of the specified parameter, in this case PV, of point TI1105 is obtained. Thus the value of TI1105.PV is used in the expression where TTEMPS(I).PV is written, when I = 1.

Note that these indirect references eliminate the need for General Inputs to get the data before the CL Block is executed, as is done in the example of Figure 3-9.

```
PACKAGE
-- Calculate the average temperature of the tank contents.
-- Include only those sensors that are covered by liquid and are not
   bad.
-- Store the average temperature as the PV.
--
CUSTOM
--
PARAMETER n  "Number of temperature sensors"
VALUE 6
--
PARAMETER ttemps: $REG_CTL ARRAY (1..6)  "Tank temperature points"
--
PARAMETER heights:  ARRAY (1..6)  "Height of each temperature sensor"
VALUE (5, 10, 15, 20, 25, 30)
--
PARAMETER level  "Tank level input"
--
PARAMETER tused: LOGICAL ARRAY (1..6)
&                                  "ON if corresponding T was used"
--
END CUSTOM
BLOCK ttcalc  (GENERIC; AT PV_ALG)
--
LOCAL tsum, nok, i
PARAMETER PVCALC
PARAMETER PVAUTOST: PVVALST
--
SET tsum = 0  -- sum of covered and OK temperature sensors
SET nok  = 0  -- number of temperatures in sum
--
sumloop: LOOP FOR i IN 1..n
IF (level > heights(i)) AND NOT BADVAL (ttemps(i).PV)
&       AND (heights(i) >= 0)
&       THEN (SET tsum = tsum + ttemps(i).PV;
&       SET nok = nok + 1;
&       SET tused(i) = ON)
ELSE  SET tused(i) = OFF
REPEAT sumloop
--
IF nok > 0  THEN SET PVCALC = tsum/nok
ELSE CALL SET_BAD (PVCALC)
--
SET PVAUTOST = NORMAL
END ttcalc
END PACKAGE
```

Figure 3-11 — Indirect Reference

The following compares the two methods of writing generic CL Blocks:

Indirect Access	General Inputs
The data point that contains the accessed data is specified when the bound data point is built; the parameter to be accessed is fixed in the program.	The data point and the parameter that contain the accessed data are specified when the bound data point is built.
From one data point name in the CDS, the program can access any number of parameters.	Each parameter that is accessed by the program requires a unique General Input.
There can be any number of data point names stored in parameters of the bound data point and used for indirect access.	There is a maximum of eight General Inputs for each data point.

A data point name can be stored into a parameter of type data point identifier:

- when a point is built or rebuilt
- through a schematic actor or Detail Display-Custom Page.
- through CG or AM-GI/GO store.
- through CL MOVE_PARAMETER and SET_NULL_POINT_ID subroutines.

It is not presently possible to do the following kind of assignment in a CL block:

```
SET usedpt = ptnames(i)
```

where USEDPT and PTNAMES are parameters of type data point identifier.

3.14.2 Type Enumeration

Enumerated data types are not just a single data type, but a class of data types. Each enumerated data type defines the particular values that a variable of that type can assume. You can define new data types within the class of enumerated data types.

An example of a definition of an enumerated data type is

```
ENUMERATION vstates = closed/stuck/open
```

VSTATES is not a variable; it is the name of a data type, just like NUMBER, LOGICAL, or TIME. The possible values that variables of type VSTATES can take on are CLOSED, STUCK, or OPEN.

The ENUMERATION statement is compiled in a package (-UL switch is required) or by itself outside of any CDS or CL Block. Once the ENUMERATION is compiled, its state names can never be changed. It can be used in PARAMETER and LOCAL declarations in any CDSs or CL Blocks, such as

```
PARAMETER feedvlv, mainvlv: vstates -- CDS or block

LOCAL feedvlv, mainvlv: vstates -- block
```

In these (mutually exclusive) declarations, FEEDVLV and MAINVLV are declared to be variables of type VSTATES; they can assume the values CLOSED, STUCK, or OPEN. If an enumerated data type is to be used for only local variables in a block and is not to be defined for global use, the following alternative declaration can be used within the block:

```
LOCAL feedvlv, mainvlv:  closed/stuck/open
```

With this method, the enumerated data type is not given a name.

The following example shows how the variables of the example enumeration type can be used after any of the above declarations are made:

```
IF PV IN lowflow + 5 .. highflow - 5 THEN SET feedvlv = stuck
ELSE IF PV < lowflow + 5 THEN SET feedvlv = closed
ELSE SET feedvlv = open
...
SET mainvlv = feedvlv
IF mainvlv = open THEN ...
```

There are many enumeration data types already built into the system. An example we have already encountered is type PVVALST, which is the data type of the parameters PVAUTOST (the status of the value stored in PVCALC) and PVSTS (the status of the value stored in PV). See Figure 3-8 and Figure 3-9. Variables of type PVVALST can assume the values NORMAL, UNCERTN and BAD.

Another built-in enumeration type is MODE. Variables of type MODE can have the values MAN/AUTO/CAS/NORMAL. The parameter MODE (control mode) is of type MODE; in this case the parameter name is the same as the name of an enumerated data type, which is permitted.

3.14.3 Type String

The value of a variable of type string is a string of any characters, up to a maximum length of 78 characters (40 characters for a CDS parameter of type string). A string constant is specified by enclosing the characters in quotation marks:

```
"This is a string."
```

If you want a quotation mark within a string, two quotation marks must be used to indicate a single quotation mark:

```
"Check Section 5 of the ""Emergency Manual"""
```

A string constant can be used to send a message to the operator through the SEND statement in a CL/AM program:

```
SEND:    "Start no. 5 conveyor belt"

IF cflag = ON THEN SEND:  "Then start no. 6 conveyor belt"
```

These messages go to all Universal Stations that have been assigned the unit of the data point to which the CL Block is attached. The messages go to a buffer in the US and the MSG key begins blinking. When the operator presses this key, all messages that have not been cleared are displayed. The operator can acknowledge and clear messages.

The following example sends the appropriate message to tell the operator how to recover when a condition occurs that a feedback controller cannot handle:

```
LOCAL instruct:  STRING
...
IF PV > PVHITP + 10 THEN
&   (SET instruct =
&        (WHEN PV = AUTO OR PV = CAS: "--put fuel controller in MAN";
&        WHEN OTHERS:  "");
&   SEND: "Furnace overheat ", instruct, "--close fuel valve")
```

If the controller is in AUTO or CAS mode, the message reads

```
Furnace overheat -- put fuel controller in MAN -- close fuel valve
```

If the controller is in MAN mode, the message reads

```
Furnace overheat -- close fuel valve
```

Expressions of any type (except enumeration constants and data-point identifiers) can appear in a SEND statement, separated by commas, and the value of each expression is displayed in its place in the message that is sent.

As with any parameter type, you can put parameters of type string on schematic displays. This allows CL Blocks to alter the text of a display, and operators can enter text into a parameter of type string that has been put on a display.

The example in Figure 3-12 shows a package that provides a scratchpad facility that allows the operators to enter any kind of messages and comments about the process operation whenever something noteworthy occurs. The program keeps the last 50 messages in a library. A display of the messages that were entered during the last specified number of hours can be requested. For example, when preparing his report at the end of the shift, the shift supervisor might request the messages that were entered during the last eight hours.

The package in Figure 3-12 works together with the user-built schematic display shown in Figure 3-13. Targets and ports on the display affect the CDS variables as follows:

- Below ENTER MESSAGE is an input port through which the operator enters his message into IN_MESG.

- ENTER MESSAGE is a target that opens the input port for message entry when it is touched, and then does a "process special" of the MESG_LIB data point when the operator enters the message. (A data point executes immediately when the standard parameter PPS of the data point is set to ON. This is often called a "process special" of the data point.) The program then puts the message IN_MESG into the library array MESSAGES and puts blanks into IN_MESG to verify that the message was saved.

- DURATION is a target that opens an input port through which the operator enters a value for DURATION.

- READ MESSAGE is a target that sets OUT_FLAG = ON and then does a "process special" of the MESG_LIB data point. The program then sends all the messages that were entered during the last DURATION number of hours and sets OUT_FLAG to OFF.

The package is implemented on a custom data point called MESG_LIB. No scheduling is specified for the data point. Because the MESG_LIB data point is not scheduled, it is executed only in response to the "process special" issued when the operator touches the ENTER MESSAGE target or the READ MESSAGE target. Because no built-in functions are provided by the system on a custom type of data point, all the actions taken when the point is executed are done by the MESG_IO block.

Two new functions are used: LEN returns the length of the string that is its argument and NOW returns seconds since midnight. We have used DATE_TIME before; it returns seconds since 1 Jan 1979. Note that DATE_TIME is used to determine if the message was entered during the last DURATION number of hours, because with DATE_TIME it is not necessary to worry about the reset of time to zero at midnight. NOW time is used so that the time-of-day of message entry can be output.

3.15 CL/AM BLOCK AS PV OR CONTROL ALGORITHM

Built-in PV and control algorithms do more than just calculate a PV or control output. As we have seen, a PV algorithm puts the value intended for the PV into PVCALC because the built-in processing functions move the value from PVCALC to PV if PVSOURCE = AUTO. A PV algorithm must also set PVAUTOST to reflect the status (NORMAL or BAD) of the PVCALC value because the built-in processing functions move PVAUTOST to PVSTS if PVSOURCE = AUTO; then PVSTS controls the display behavior of the PV. If you write a CL block to take the place of a built-in PV algorithm, your CL block needs to work with these standard parameters in the same way that a built-in algorithm does. For more information, see the CL PV Algorithm section of the *Application Module Algorithm Engineering Data.*

As with the PV algorithms, there are a number of standard parameters and conventions that are used by the built-in control algorithms to interface with built-in processing functions for such things as back initialization, antiwindup-status propagation, and control mode rules. If you write a CL block to take the place of a built-in control algorithm, your CL block may need to work with some of these standard parameters and should follow the conventions and philosophies of the built-in functions where applicable. See the CL Control Algorithm section of the *Application Module Algorithm Engineering Data* for information that you will need to write a CL block that will be used as a control algorithm.

```
PACKAGE
-- Maintain a library of messages and comments concerning the process operation.
-- These messages may be entered by the operator and shift supervisor.
--
-- When requested, output all messages that were entered during the last specified
-- number of hours.

CUSTOM (NOT BLD_VISIBLE)
--
PARAMETER max: NUMBER "Max messages in library"
VALUE 50
--
PARAMETER ptr: NUMBER "Pointer to last message"
VALUE 50
--
PARAMETER messages: STRING ARRAY (1..50) "Stored messages"
--
PARAMETER mes_time: TIME ARRAY (1..50) "Date/time of message"
--
PARAMETER out_time: TIME ARRAY (1..50) "Time of message"
--
PARAMETER out_flag: LOGICAL "Set ON to output messages"
VALUE OFF
--
PARAMETER in_mesg: STRING "Entered message"
--
PARAMETER duration: NUMBER "Specified no. of hours"
--
END CUSTOM
--
BLOCK mesg_io (POINT mesg_lib; AT GENERAL)
--
LOCAL i, count
--
--  If a new message was entered, put it in library.
IF in_mesg = " " THEN GOTO bymesg              -- Bypass if no new message
SET ptr = ptr + 1                              -- Increment storage pointer
IF ptr > max THEN SET ptr = 1                  -- Wrap around
SET messages(ptr) = in_mesg                    -- Store new message over oldest
SET in_mesg = " "                              -- Erase input on display
SET mes_time(ptr) = DATE TIME                  -- Save system time for check
SET out_time(ptr) = NOW                        -- Save time of day for output
--
--  If a request was made, output the messages.
bymesg: IF out_flag = OFF THEN EXIT            -- Bypass if no request
SEND: "Messages during last ", duration, " hours"      -- Header
SET i = ptr + 1                                -- Start with most recent message
--
rep: LOOP FOR count IN 1..max
SET i = i - 1                                  -- Next oldest
IF i < 1 THEN SET i = max                      -- Wrap around
IF mes_time(i) >= DATE_TIME - duration HOURS THEN
&      (SEND: out_time(i), "  ", messages(i);
&      REPEAT rep)
--
SEND: "Message output is complete"
SET out_flag = OFF
--
END mesg_io
END PACKAGE
```

Figure 3-12 — Message Library Package

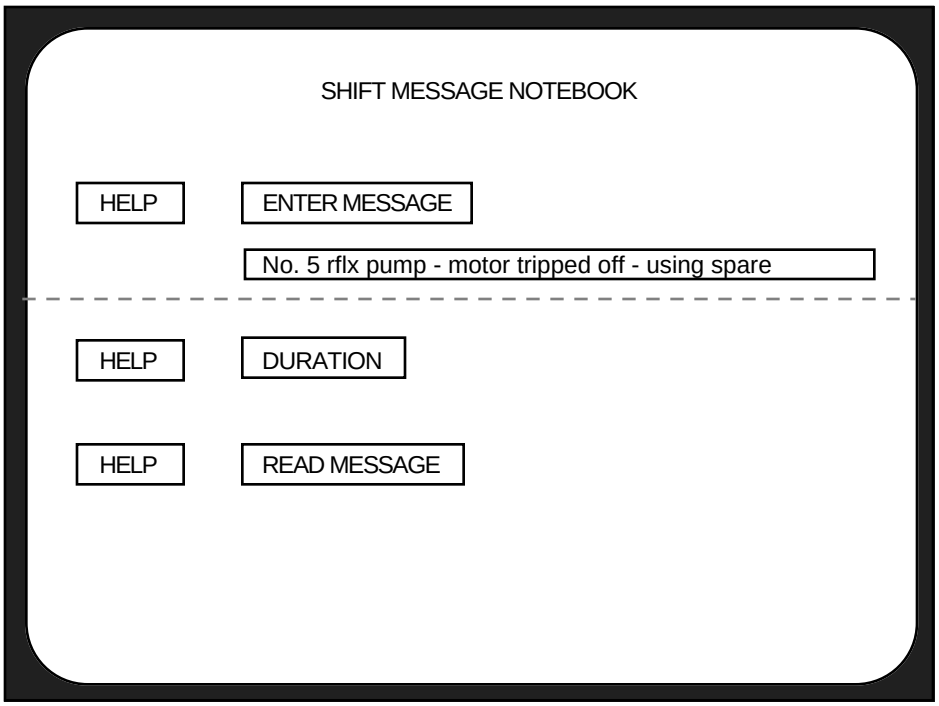


Figure 3-13 — Message Library Schematic Display 2805

Index

Topic	Section Heading
&	3.5.2
Abort, program	3.5.2
Activation of Data Point	3.9
Addressing of Data Points	3.2, 3.5.2
Algorithms	
built-in	3.3
replacement by CL Programs	3.4, 3.15
AM Database	3.2
Arrays of values	3.12.2
Attributes of Parameters	3.11
Bad Value	3.5.2
Block, defined	3.5.2
Block—see also CL Program	
Bound Data Point	3.5.2
Building a Data Point	3.5.1
Built-In Algorithms	3.3
Built-in Functions (LEN, NOW)	3.14.3
Built-in Functions	3.12.2
Built-in Subroutines	3.12.2
CDS 3.11	
CL Block defined	3.5.2
CL Program	
as standard control algorithm replacement	3.4
as standard PV algorithm replacement	3.4
comments in	3.5.2
examples	3.5
Insertion point	3.4
linked to data point	3.4
Comments in CL programs	3.5.2
Compilation of program	3.7
Continuation lines	3.5.2
Control Algorithm, replacement by CL Block	3.15
Control Strategies	2.0
CTLALGID parameter	3.3
Custom Data Segments	3.11
Database, AM	3.2
Data Entity Builder	3.5.1
Data Point	
Activation	3.9
building	3.5.1
Displays	3.5.2
Parameters	3.2, 3.5.2
Bound	3.5.2
CL Program linked to	3.4
description	3.2
parameters appended to	3.3
Data Points	
and CL execution	3.4
access to	3.5.2
Data Type, default value	3.11

Index

Topic	Section Heading
Data Types	
Data-Point Identifier	3.14.1
Enumeration	3.14.2
Logical	3.12.2
Number	3.12.2
String	3.14.3
Time	3.12.3
Data-Point Identifier Data Type	3.14.1
DATE_TIME built-in function	3.14.3
DEB 3.5.1	
Declarations	
EXTERNAL	3.5.2
LOCAL	3.5.2
PARAMETER	3.5.2
Displays, Data Point	3.5.2
Dot notation in Data Point addressing	3.2, 3.5.2
Enumeration Data Type	3.14.2
EU Attribute of Parameter	3.11
Execution modes	3.4
EXTERNAL declaration	3.5.2
File Concepts	3.6
Functions, built-in	3.12.2
General Inputs	3.12.1
Generic Blocks, two methods	3.14.1
Generic Package example	3.12.1
Identifiers	3.5.2
IF THEN ELSE statement	3.12.2
Indirect Reference to Data Points	3.14.1
Insertion points for CL Programs	3.4, Table 3-1
Installation, summary of Program	3.10
LEN built-in Function	3.14.3
Library File, System	3.11
Linking	3.8
LOCAL declaration	3.5.2
LOGICAL Data type	3.12.2
LOOP statement	3.12.2
MC	2.0
Multifunction Controller	2.0
NOW built in Function	3.14.3
Number Data Type	3.12.2
PACKAGE	3.12.1
PARAMETER declaration	3.1, 3.5.2
Parameter Entry Display	3.5.1
Parameter Lists	3.13
Parameters	
description	3.2
appended to data points	3.3
of Data Points	3.2, 3.5.2
on Data Points, addressing	3.5.2

Index

Topic	Section Heading
PED 3.5.1	
PM	2.0
Process Manager	2.0
Process Unit	3.5.1
Program	
Abort	3.5.2
Compilation	3.7
Installation, summary of	3.10
Linking	3.8
source entry	3.6.1
Statements	3.5.2
Generic	3.12.1
Program—see also CL Block	
PV Algorithm, replacement by CL Block	3.15
PVALGID parameter	3.3
Source entry, program	3.6.1
Statements, Program	3.5.2
Status Displays	3.5.2
String Data Type	3.14.3
Subroutines, built-in	3.12.2
System Library File	3.11
Time Data Type	3.12.3
UNIT 3.5.1	
VALUE Attribute of Parameter	3.11
Value Arrays	3.12.2
Value, Bad	3.5.2
Variables, types of	3.12.2
Variables—see also Data Types	

READER COMMENTS

Honeywell IAC Automation College welcomes your comments and suggestions to improve future editions of this and other publications.

You can communicate your thoughts to us by fax, mail, or toll-free telephone call. We would like to acknowledge your comments; please include your complet name and address

BY FAX: Use this form; and fax to us at (602) 313-4108

BY TELEPHONE: In the U.S.A. use our toll-free number 1*800-822-7673 (available in the 48 contiguous states except Arizona; in Arizona dial 1-602-313-5558).

BY MAIL: Use this form; detach, fold, tape closed, and mail to us.

Title of Publication: **Control Language Application Module Overview** Issue Date: **9/95**

Publication Number: **SW27-500**

Writer: **Maria Nelson**

COMMENTS: _____

_____

_____

_____

_____

_____

_____

_____

RECOMMENDATIONS:_____

_____

_____

_____

_____

_____

_____

_____

NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

TELEPHONE _____ FAX _____

(If returning by mail, please tape closed; Postal regulations prohibit use of staples.)

Communications concerning technical publications should be directed to:

Automation College
Industrial Automation and Control
Honeywell Inc.
2820 West Kelton Lane
Phoenix, Arizona 85023-3028

FOLD FOLD

From: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE USA

BUSINESS REPLY MAIL

FIRST CLASS **PERMIT NO. 4332** PHOENIX, ARIZONA

POSTAGE WILL BE PAID BY ....

Honeywell

Industrial Automation and Control
2820 West Kelton Lane
Phoenix, Arizona 85023-3028

Attention: Manager, Quality

FOLD FOLD

Cut Along Line

Additional Comments:

Honeywell

Industrial Automation and Control
Honeywell Inc.
16404 North Black Canyon Highway
Phoenix, Arizona 85023-3099

Helping You Control Your World