C2000 Digital Control Library

Version 3.1

User's Guide

October 2018

Read This First

About This Manual

This User's Guide covers version 3.1 of the C2000 Digital Control Library (DCL). It contains technical descriptions of the library functions and how to use them. The DCL User's Guide does not cover control applications, control theory, or technical details of C2000 devices; however some information on these topics may be found in the references listed in chapter 6.

The User's Guide is divided into six chapters. Chapter 1 presents a general introduction to the library and provides background information. Chapter 2 describes the library contents and structure, how to add library functions to a user's program, and how to migrate existing source code to be compatible with version 3.0. Chapter 3 describes the controllers and supporting functions, and provides information on their use. Chapter 4 describes the utilities supplied with the library, including data loggers, a transient capture module, and simulation models. Chapter 5 describes a set of supporting software examples which illustrate the use of the library. A list of relevant technical references and training can be found in Chapter 6.

How to Use This Manual

New users are advised to begin by reading the library overview in chapter 1. Chapter 2 provides a useful step-by-step guide of how to integrate the library into an existing C program as well as how to migrate from earlier versions, and should be read carefully by all users. Once the user has decided on the type of controller to implement, performance and other important information in the relevant sub-section of chapter 3 should be read carefully. If data array management or performance measurement is required, the utilities described in chapter 4 may be of interest. The library includes a number of software examples which illustrate the use of most of the controllers. These are described in chapter 5 and should serve as a good starting point for new users. Finally, for further reading and other resources, users are directed to the list of technical references and training materials in chapter 6.

The library is supplied entirely in the form of C and assembly source code. There is no object code in the library. Users who are familiar with C may find it helpful to refer to the source code while reading the function descriptions in chapters 3 and 4.

Related Documentation from Texas Instruments

Technical documentation and development tools for the C2000 device can be found online on the Texas Instruments website at www.ti.com/c2000.

If You Need Assistance

Technical support for C2000 products is available online via the TI “E2E” Community: e2e.ti.com/support/microcontrollers/c2000

Contents

Read This First ..... **iii**
About This Manual iii
How to Use This Manual iii
Related Documentation from Texas Instruments iii
If You Need Assistance iv
Contents ..... **v**
Figures ..... **xii**
Tables ..... **xiii**
Introduction **1**
1.1 Supported Devices 1
1.2 Overview of the Library 2
1.3 New In This Version 3
1.3.1 New Features 3
1.3.2 Function Naming 4
1.3.3 Structure Naming 5
1.3.4 Calling Convention 5
1.3.5 Data Types 5
1.3.6 The ZPK3 Structure 5
1.3.7 Compensator Stability Tests 6
1.3.8 Bug Fixes 6
1.4 CPU Compatibility 6
1.5 Benchmarks 7
Using the Digital Control Library **12**
2.1 What the Library Contains 12
2.1.1 Header Files 12
2.1.2 Source Files 13
2.1.3 Examples 14
2.2 Header File Dependency 14
2.3 How to Add the DCL to User Code 15
2.3.1 Steps to Add the DCL to Existing C Code 16
2.3.2 Calling the Library Functions from Assembly 19
2.4 Updating Controller Parameters 19
2.5 Error Handling 21
2.6 How to Modify the Library Code 22
Controllers **24**
3.1 Linear PID Controllers 26
3.1.1 Description 26
3.1.2 Implementation 28
3.1.3 Functions 30
DCL_runPID_C1 30
DCL_runPID_C2 30
DCL_runPID_C3 31
DCL_runPID_C4 31
DCL_runPID_L1 32

DCL_runPID_L2 32
DCL_resetPID 32
DCL_updatePID 33
DCL_fupdatePID 33
DCL_setPIDfilterBW 33
DCL_setActivePIDfilterBW 34
DCL_getPIDfilterBW 34
DCL_loadSeriesPIDasZPK 34
DCL_loadParallelPIDasZPK 35
3.2 Linear PI Controllers 35
3.2.1 Description 35
3.2.2 Implementation 36
3.2.3 Functions 37
DCL_runPI_C1 37
DCL_runPI_C2 38
DCL_runPI_C3 38
DCL_runPI_C4 38
DCL_runPI_C5 39
DCL_runPI_L1 39
DCL_runPI_L2 39
DCL_runPI_L3 40
DCL_runPI_L4 40
DCL_resetPI 41
DCL_updatePI 41
DCL_fupdatePI 41
DCL_loadSeriesPIasZPK 42
DCL_loadParallelPIasZPK 42
3.3 Non-linear PID Controller 42
3.3.1 Description 42
3.3.2 Implementation 46
3.3.3 Functions 47
DCL_runNLPID_C1 47
DCL_runNLPID_C2 47
DCL_setGamma 48
DCL_resetNLPID 48
DCL_updateNLPID 48
DCL_setActiveNLPIDfilterBW 49
DCL_setNLPIDfilterBW 49
DCL_getNLPIDfilterBW 49
DCL_getNLPIDgamma 50
DCL_getNLPIDdelta 50
3.4 Non-linear PI Controllers 50
3.4.1 Description 50
3.4.2 Implementation 50
3.4.3 Functions 51
DCL_runNLPI_C1 51
DCL_resetNLPI 51
DCL_updateNLPI 52
3.5 Double Integrator PI Controller 52
3.5.1 Description 52
3.5.2 Implementation 52
3.5.3 Functions 53

DCL_runPI2_C1 53
DCL_resetPI2 53
DCL_updatePI2 54
DCL_fupdatePI2 54
3.6 Direct Form 1 (First Order) Compensators 54
3.6.1 Description 54
3.6.2 Implementation 55
3.6.3 Functions 55
DCL_runDF11_C1 55
DCL_runDF11_C2 56
DCL_runDF11_L1 56
DCL_resetDF11 56
DCL_updateDF11 57
DCL_fupdateDF11 57
DCL_isStableDF11 57
DCL_loadDF11asZPK 58
DCL_loadDF11asPI 58
3.7 Direct Form 1 (Third Order) Compensators 59
3.7.1 Description 59
3.7.2 Implementation 61
3.7.3 Functions 62
DCL_runDF13_C1 62
DCL_runDF13_C2 62
DCL_runDF13_C3 63
DCL_runDF13_C4 63
DCL_runDF13_C5 63
DCL_runDF13_C6 64
DCL_runDF13_L1 64
DCL_runDF13_L2 64
DCL_runDF13_L3 65
DCL_resetDF13 65
DCL_updateDF13 65
DCL_fupdateDF13 66
DCL_isStableDF13 66
DCL_loadDF13asZPK 66
3.8 Direct Form 2 (Second Order) Compensators 67
3.8.1 Description 67
3.8.2 Implementation 69
3.8.3 Functions 69
DCL_runDF22_C1 69
DCL_runDF22_C2 70
DCL_runDF22_C3 70
DCL_runDF22_C4 70
DCL_runDF22_C5 71
DCL_runDF22_C6 71
DCL_runDF22_L1 71
DCL_runDF22_L2 72
DCL_runDF22_L3 72
DCL_runDF22_L4 72
DCL_resetDF22 73
DCL_updateDF22 73
DCL_fupdateDF22 73

DCL_isStableDF22 74
DCL_loadDF22asZPK 74
DCL_loadDF22asParallelPID 74
DCL_loadDF22asSeriesPID 75
3.9 Direct Form 2 (Third Order) Compensators 75
3.9.1 Description 75
3.9.2 Implementation 77
3.9.3 Functions 77
DCL_runDF23_C1 77
DCL_runDF23_C2 78
DCL_runDF23_C3 78
DCL_runDF23_C4 78
DCL_runDF23_C5 79
DCL_runDF23_C6 79
DCL_runDF23_L1 79
DCL_runDF23_L2 80
DCL_runDF23_L3 80
DCL_resetDF23 80
DCL_updateDF23 81
DCL_fupdateDF23 81
DCL_isStableDF23 81
DCL_loadDF23asZPK 82
3.10 Fixed-Point PID Controllers 82
3.10.1 Description 82
3.10.2 Implementation 82
3.10.3 Functions 84
DCL_runPID_A1 84
DCL_resetPID32 85
DCL_updatePID32 85
DCL_fupdatePID32 85
3.11 Fixed-Point PI Controllers 86
3.11.1 Description 86
3.11.2 Implementation 86
3.11.3 Functions 87
DCL_runPI_A1 87
DCL_resetPI32 87
DCL_updatePI32 88
DCL_fupdatePI32 88
3.12 Gain Scheduler Module 89
3.12.1 Description 89
3.12.2 Implementation 89
3.12.3 Functions 90
DCL_runGSM_C1 90
DCL_resetGSM 90
DCL_updateGSM 90
DCL_fupdateGSM 91
DCL_loadGSMoffsets 91
DCL_loadGSMgains 91
3.13 Non-linear Control Law 92
3.13.1 Description 92
3.13.2 Implementation 92
3.13.3 Functions 92

DCL_runNLF_C1 92
3.14 Double Precision PID Controllers 93
3.14.1 Description 93
3.14.2 Implementation 93
3.14.3 Functions 93
DCL_runPIDF64_S1 93
DCL_resetPIDF64 93
DCL_updatePIDF64 94
DCL_setPIDF64filterBW 94
DCL_setActivePIDF64filterBW 94
DCL_getPIDF64filterBW 95
Utilities 96
4.1 Control Clamps 96
4.1.1 Description 96
4.1.2 Functions 97
DCL_runClamp_C1 97
DCL_runClamp_C2 97
DCL_runClamp_L1 98
4.2 Floating Point Data Logging Functions 98
4.2.1 Description 98
4.2.2 Functions 100
DCL_deleteLog 100
DCL_resetLog 100
DCL_initLog 100
DCL_writeLog 101
DCL_fillLog 101
DCL_clearLog 101
DCL_readLog 102
DCL_copyLog 102
DCL_freadLog 102
DCL_fwriteLog 102
4.3 4-channel Floating Point Data Logger 103
4.3.1 Description 103
4.3.2 Functions 104
DCL_initMLOG 104
DCL_resetMLOG 104
DCL_armMLOG 105
DCL_runMLOG 105
4.4 Transient Capture Module 105
4.4.1 TCM_idle Mode 106
4.4.2 TCM_armed Mode 107
4.4.3 TCM_capture Mode 108
4.4.4 TCM_complete Mode 109
4.4.5 Functions 110
DCL_initTCM 110
DCL_resetTCM 110
DCL_armTCM 110
DCL_runTCM 111
4.5 Performance Measurement 111
4.5.1 Description 111
4.5.2 Functions 112
DCL_runIES_C1 112

DCL_runIES_C2 113
DCL_runIAE_C1 113
DCL_runIAE_C2 113
DCL_runITAE_C1 113
DCL_runITAE_C2 114
4.6 Fixed Point Data Logger Support 114
4.6.1 Description 114
4.6.2 Functions 114
DCL_deleteLog32 114
DCL_resetLog32 115
DCL_initLog32 115
DCL_writeLog32 115
DCL_fillLog32 116
DCL_clearLog32 116
DCL_readLog32 116
DCL_copyLog32 116
DCL_initMLOG32 117
DCL_resetMLOG32 117
DCL_armMLOG32 118
DCL_runMLOG32 118
4.7 Simulation Models 118
4.7.1 The DCL Block-set 118
4.7.2 Simulation Example 119
4.8 Double Precision Data Logging Functions 119
4.8.1 Description 119
4.8.2 Functions 121
DCL_deleteLog64 121
DCL_resetLog64 121
DCL_initLog64 121
DCL_writeLog64 122
DCL_fillLog64 122
DCL_clearLog64 122
DCL_readLog64 123
DCL_copyLog64 123
Examples 124
5.1 Example 1: DF22 Compensator Running on FPU32 125
5.1.1 Example Overview 125
5.1.2 Code Description 125
5.1.3 Running the Example 126
5.2 Example 2: DF23 Compensator Running on CLA 127
5.2.1 Example Overview 127
5.2.2 Code Description 127
5.2.3 Running the Example 128
5.3 Example 3: NLPID Controller Running on FPU32 129
5.3.1 Example Overview 129
5.3.2 Code Description 129
5.3.3 Running the Example 130
5.4 Example 4: PI Controller Running on CLA 131
5.4.1 Example Overview 131
5.4.2 Code Description 131
5.4.3 Running the Example 131
5.5 Example 5: PID Controller Running on FPU32 132

5.5.1 Example Overview 132
5.5.2 Code Description 132
5.5.3 Running the Example 133
5.6 Example 6: TCM Running on FPU32 133
5.6.1 Example Overview 133
5.6.2 Code Description 134
5.6.3 Running the Example 134
5.7 Example 7: Smith Predictor Running on FPU32 135
5.7.1 Example Overview 135
5.7.2 Code Description 136
5.7.3 Running the Example 136
5.8 Example 8: GSM Running on FPU32 136
5.8.1 Example Overview 136
5.8.2 Code Description 137
5.8.3 Running the Example 137
5.9 Example 9: Multiple Controller System with ERAD Running on FPU32 138
5.9.1 Example Overview 138
5.9.2 Code Description 140
5.9.3 Running the Example 140
Support 141
6.1 References 141
6.1.1 C2000 Documentation 141
6.1.2 Literature 141
6.2 Training 142
6.3 Support 142

Figures

Figure 1. DCL function naming 4
Figure 2. DCL Header File Dependency 15
Figure 3. DCL Data Logger Header Dependency 15
Figure 4. CCSv6 Include Options 17
Figure 5. Parallel form PID controller 26
Figure 6. PID control action 27
Figure 7. DCL_PID C1, C2, & L1 architecture 29
Figure 8. DCL_PID C3, C4, & L2 architecture 30
Figure 9. DCL_PI C1, C2, L1, & L3 architecture 36
Figure 10. DCL_PI C3, C4, L2 & L4 architecture 36
Figure 11. DCL_PI C5 architecture 37
Figure 12. DCL_NLPID C1 input architecture 43
Figure 13. DCL_NLPID C1 output architecture 44
Figure 14. DCL_NLPID C2 output architecture 44
Figure 15. Non-linear control law input-output plot 44
Figure 16. Gain vs error curves for varying alpha 45
Figure 17. NLPID linearized region 46
Figure 18. The DCL_NLPI_C1 architecture 51
Figure 19. DCL_PI2 C1 architecture 53
Figure 20. DCL_DF11 C1, C2, & L1 architecture 55
Figure 21. DCL_DF13 C1, C4, & L1 architecture 60
Figure 22. DCL_DF13 C2, C3, C5, C6, L2, & L3 architecture 61
Figure 23. DCL_DF13 data & coefficient layout 62
Figure 24. DCL_DF22 C1, C4,L1, & L4 architecture 68
Figure 25. DCL_DF22 C2, C5, & L2 architecture 68
Figure 26. DCL_DF22 C3, C6, & L3 architecture 69
Figure 27. DCL_DF23 C1, C4, & L1 architecture 76
Figure 28. DCL_DF23 C2, C5, & L2 architecture 76
Figure 29. DCL_DF23 C3, C6, & L3 architecture 77
Figure 30. DCL_PID32 A1 architecture 84
Figure 31. DCL_PI A1 architecture 87
Figure 32. DCL_GSM sector numbering 89
Figure 33. Data log pointer allocation 99
Figure 34. MLOG architecture 103
Figure 35. TCM operation in TCM_idle mode 107
Figure 36. TCM operation in TCM_armed mode 108
Figure 37. TCM operation in capture mode (monitor frame un-winding) 108
Figure 38. TCM operation in TCM_capture mode (lead frame complete) 109
Figure 39. TCM capture complete 109
Figure 40. Transient servo error 111
Figure 41. FDLOG64 pointer allocation 120
Figure 42. Smith Predictor control loop 136

Tables

Table 1.	CPU compatibility - controllers	6
Table 2.	CPU compatibility – support modules	7
Table 3.	Controller execution & code size benchmarks	7
Table 4.	Support function execution benchmarks	9
Table 5.	Fast update function execution benchmarks	11
Table 6.	List of DCL header files	12
Table 7.	List of DCL source files	13
Table 8.	List of DCL code examples	14
Table 9.	List of fast parameter update function execution cycles	20
Table 10.	List of CSS enumerated error codes	21
Table 11.	Data log read/write benchmarks	99
Table 12.	Performance index function benchmarks	112
Table 13.	FDLOG64 read/write benchmarks	121

Chapter 1

Introduction

This chapter contains a brief introduction to the Texas Instruments C2000 Digital Control Library.

Section

1.1 Supported Devices

1.2 Overview of the Library

1.3 New in this Version

1.4 Benchmarks

1.1 Supported Devices

The DCL is compatible with three different C2000 CPUs:

- A 32-bit floating point CPU, denoted “FPU32”
- A 32-bit floating point “Control Law Accelerator”, denoted “CLA”
- A 32-bit fixed point CPU, denoted “C28x”

Among those devices which contain an FPU32 are:

- TMS320F28004x
- TMS320F2837xD
- TMS320F2837xS
- TMS320F2807x
- TMS320F2833x
- TMS320C2834x
- TMS320F2806x
- TMS320F28M35x
- TMS320F28M36x

The library also includes functions optimized for use on the Control Law Accelerator (CLA). This CPU is only found on certain C2000 devices, including:

- TMS320F28004x
- TMS320F2837xD

- TMS320F2837xS
- TMS320F2807x
- TMS320F2806x
- TMS320F2805x
- TMS320F2803x

The DCL includes limited support for fixed-point C28x platforms in the form of two controllers and two data logging modules. Among the devices supported in this way are:

- TMS320F2805x
- TMS320F2804x
- TMS320F2803x
- TMS320F2802x
- TMS320F281x
- TMS320F280x

Note that the fixed-point (C28x) library functions will also run un-modified on any device which contains an FPU32. The DCL does not support the C24x CPU.

The C28x Run Time Support (RTS) library allows FPU32 functions written in C to be run on the fixed point C28x, however RTS emulation of the floating point data type is not cycle efficient and the library has not been tested in this way. Users are advised to run only those controller and CPU combinations recommended in this User's Guide. A list of controllers and compatible CPUs can be found in Table 1.

1.2 Overview of the Library

The C2000 Digital Control Library (DCL) provides a suite of robust software functions for developers of digital control applications using the Texas Instruments C2000 MCU platform. All the functions in the library are supplied in the form of C or assembly source code. The library is delivered in the C2000Ware software suite, which is available for free download at: www.ti.com/tool/c2000ware.

The DCL functions are intended for use in any system in which a C2000 device is used. The DCL may not be used with any other devices. Refer to the C2000Ware license agreement for further information. The DCL is independent of other application specific C2000 software libraries and Software Development Kits (SDKs), however providing attention is paid to data type and numerical range, integration with those packages is straightforward.

Version 3.1 of the DCL contains 169 controller and supporting functions. Ten different types of controller are represented: five PID types, and five "Direct Form" types. The former are typically used to tune properties of a transient response, while the latter are typically used to shape the open loop frequency response. Version v3.x of the DCL contains functions to convert controller parameters from one type to the other; for example, the user may emulate PID control using a direct form 2 controller structure. The library also contains linear and non-linear gain schedulers.

Supporting functions fall into three groups: data logging, performance measurement, and transient capture. All support functions run on the C28x or FPU32 only (i.e. they not

compatible with the CLA). Most are supplied in C code form; however a small number of time-critical functions are also supplied as C callable assembly functions.

The library includes a set of example projects which illustrate how DCL functions might be applied in a user project. All examples were compiled for the F28069 or F280049 devices and are based on the TI peripheral register header files. The library contains no device specific code and users of other C2000 devices will find it straightforward to apply the examples to their own projects.

The DCL does not contain tools to measure frequency response or perform compensator parameter selection; however similar features can be found in the "Compensation Designer" utility which is part of the TI "powerSUITE" package. A Software Frequency Response Analyzer (SFRA) utility can be found at: www.ti.com/tool/sfra.

1.3 New In This Version

1.3.1 New Features

Version 3.1 of the DCL adds the following new features:

- Series & parallel PI controllers for the CLA coded in C
- A full DF22 compensator for the CLA coded in C

Version 3.0 of the DCL added the following new features:

- Reset functions which clear stored data without modifying other parameters
- Functions to perform safe controller parameter updates
- Error checking capability
- Functions to test stability of direct form compensators
- Ability to enter controller parameters in the form of complex zeros and poles
- PI and PID emulation using direct form compensators
- A 64-bit floating point PID controller
- A double precision data logger
- A single precision gain scheduler module
- Non-linear PI & PID controllers
- User 'test-points' to log an internal variable for debugging and test purposes
- Sample period stored in each controller structure
- A small Simulink library and example model
- Three additional CCS example projects

From v3.0 each controller structure has been appended with two 32-bit pointers, each of which holds the address of a separate support sub-structure.

The first such sub-structure contains the Shadow Parameter Set (SPS), which the user loads prior to executing a parameter update sequence. The update sequence performs a safe copy of the SPS parameters into the main controller structure by disabling interrupts before the copy and re-enabling them afterwards. This ensures the controller never runs with a partially updated parameter set, and allows controller parameters to be updated without disturbing the control loop.

The second sub-structure is a Common Support Structure (CSS). The CSS contains supporting data used for error checking and parameter updates.

The use of SPS and CSS is optional. Users who do not need the additional features provided by SPS and CSS may safely ignore these structure elements. No changes have been made to any controller code between v2.1.1 and v3.1.

1.3.2 Function Naming

The controller function names are the same as version 2.0 and later. The naming convention allows several controllers of similar type, but with different implementations, to be used together in the same program without conflict. An example of a library function name, together with a description of the constituent fields, is shown below.

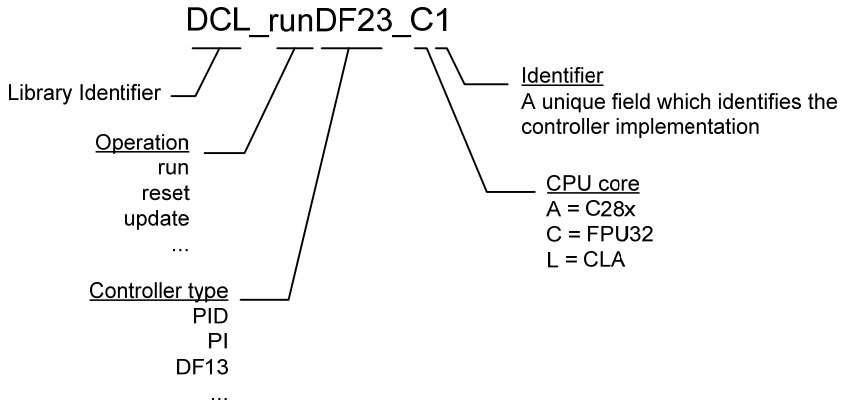


Figure 1. DCL function naming

In this format, both the controller type and the CPU on which it runs are explicit. This allows future expansion of the library with variations of each controller type. An example is the PID controller, which exists in both “ideal” and “parallel” forms in the library. The final digit is an arbitrary number which identifies the implementation. Users may add their own controller variations to the DCL controllers by changing the final two characters of the function name. Refer to section 2.6 for further information.

Version 1.0 function names have been deprecated from version 3.0 of the DCL.

In addition to “run”, all controllers now include the functions “reset” and “update”. The first resets all internal non-parametric data to its default values without changing any controller parameters. The second performs a safe parameter update without changing any non-parametric control data. Note that the “update” function requires both SPS and CSS structures. The same “verb-noun” function name construction shown in Figure 1 applies to all supporting functions.

The DCL includes on implementation of a 64-bit floating point PID controller. This controller uses the letter “S” as the CPU identifier to distinguish it from the similar FPU32 controllers. It may be compiled and run on an FPU32, however without hardware support for the double precision data type relatively poor cycle performance should be expected.

Version 3.x also contains a number of new functions intended to enhance library ease-of-use. Refer to the function descriptions in chapter 3 for more information.

1.3.3 Structure Naming

The controller structure naming convention in v3.x is broadly similar to v2.1 of the library; however the structure names of controllers which run on the CLA have been changed. Each controller structure name is prefixed by the string “DCL_”. This has been done for compatibility with other TI libraries, and avoids a common naming conflict with the mathematical constant “PI”. Note that the mapping of legacy v1.x structure names has been deprecated from v3.0 onwards.

1.3.4 Calling Convention

All functions in the DCL are designed to be called from a C program. The context save & restore in each function assumes the standard parent register save is performed be the compiler. If any of the assembly functions are called from an assembly program, additional context save & restore instructions must be added. See section 2.2.2 for further details.

1.3.5 Data Types

All library floating-point variables are declared with the `float32_t` data type. Unsigned and signed integers are declared as `uint16_t` and `int16_t` respectively. Fixed-point controllers use the signed long integer type `int32_t`. Data type definitions can be found in the `DCL.h` header file. The DCL is fully compatible with both COFF and EABI compilers.

1.3.6 The ZPK3 Structure

The library header file `DCL.h` contains a type definition of a third order transfer function in the form of zero and pole frequencies, and a real gain. This “ZPK3” structure enables direct form compensators to be configured from a frequency specification. Functions also exist to load PI and PID controllers in the same way. Refer to chapter 3 for more information on controller configuration using ZPK3.

The general form of the linear third order DT transfer function is

Equation 1. $$F(z) = K\frac{(z-q1)(z-q2)(z-q3)}{(z-p1)(z-p2)(z-p3)}$$

In Equation 1, q1, q2, and q3, represent the frequencies of the three zeros in Hz; p1, p2, and p3 represent the frequencies of the three poles in Hz; and K is the real gain.

The ZPK3 structure is also used in the library to represent first and second order transfer functions. In each case, poles and zeros on the right being ignored. For example, in a 1-pole, 1-zero description, only K, q1, and p1 are relevant; the user function will ignore q2, q3, p2, & p3.

The DCL functions which take ZPK3 arguments allow complex poles and zeros providing they exist in conjugate pairs, thereby resulting in real polynomial coefficients. Error checking is built into those functions to ensure this is always the case.

1.3.7 Compensator Stability Tests

The library contains functions to test stability of all direct form compensators. Stability is determined from the pole locations based on the transfer function denominator coefficients. Generic polynomial stability functions are found in the library header file `DCL.h`. In the first order case, determination is trivial. For second and third order compensators, the Jury array method is used. All stability test functions return the `bool` data type: 'true' if all compensator poles lie within the unit circle, otherwise 'false'.

1.3.8 Bug Fixes

No bugs were reported in version 3.0 of the DCL.

1.4 CPU Compatibility

The tables below list the CPU compatibility of the controller and support structures in the DCL.

Table 1. CPU compatibility - controllers

Controller	C28x	FPU32	CLA
DCL_PID	No	Yes	No
DCL_PI	No	Yes	No
DCL_PI2	No	Yes	No
DCL_NLPID	No	Yes	No
DCL_NLPI	No	Yes	No
DCL_DF11	No	Yes	No
DCL_DF13	No	Yes	No
DCL_DF22	No	Yes	No
DCL_DF23	No	Yes	No
DCL_GSM	No	Yes	No
DCL_PID_CLA	No	No	Yes
DCL_PI_CLA	No	No	Yes
DCL_DF11_CLA	No	No	Yes
DCL_DF13_CLA	No	No	Yes

DCL_DF22_CLA	No	No	Yes
DCL_DF23_CLA	No	No	Yes
DCL_PID32	Yes	No	No
DCL_PI23	Yes	No	No
DCL_PIDF64	No	Yes	No

Table 2. CPU compatibility – support modules

Controller	C28x	FPU32	CLA
FDLOG	No	Yes	No
MLOG	No	Yes	No
TCM	No	Yes	No
FDLOG32	Yes	No	No
MLOG32	Yes	No	No
FDLOG64	No	Yes	No

1.5 Benchmarks

The following table lists the performance of each library function by cycle count. In all cases, cycle count benchmarks were measured by logging a free-running PWM timer before and after each function call. Therefore the measured cycle count includes the function calling overhead from the C environment. In cases where the cycle count is dependent on input data, such as in anti-windup logic, the maximum figure is always given. Compiler optimization was disabled in all tests. Function sizes are given in units of 16-bit words, as reported in the ".map" file.

Table 3. Controller execution & code size benchmarks

Function	Cycles	Size (W)
DCL_runPID_A1	99	80
DCL_runPID_C1	83	99
DCL_runPID_C2	197	207
DCL_runPID_C3	186	196
DCL_runPID_C4	86	92
DCL_runPID_L1	53	70
DCL_runPID_L2	45	58
DCL_runPI_A1	62	46

DCL_runPI_C1	52	54
DCL_runPI_C2	117	121
DCL_runPI_C3	122	126
DCL_runPI_C4	48	37
DCL_runPI_C5	194	180
DCL_runPI_L1	34	42
DCL_runPI_L2	33	40
DCL_runPI_L3	118	238
DCL_runPI_L4	122	248
DCL_runNLPID_C1	284[(1)(2)]	312
DCL_setGamma	2090[(2)]	
DCL_runNLPID_C2	3353[(1)]	3297[(1)]
DCL_runNLPI	2230[(1)]	2197[(1)]
DCL_runPI2_C1	218	201
DCL_runDF11_C1	37	23
DCL_runDF11_C2	60	
DCL_runDF11_L1	30	34
DCL_runDF13_C1	71	66
DCL_runDF13_C2	20	79
DCL_runDF13_C3	74	
DCL_runDF13_C4	175	162
DCL_runDF13_C5	40	38
DCL_runDF13_C6	121	126
DCL_runDF13_L1	61	86
DCL_runDF13_L2	20	100
DCL_runDF13_L3	58	
DCL_runDF22_C1	44	45
DCL_runDF22_C2	19	48
DCL_runDF22_C3	39	
DCL_runDF22_C4	71	75
DCL_runDF22_C5	29	26
DCL_runDF22_C6	60	67
DCL_runDF22_L1	33	40
DCL_runDF22_L2	20	60
DCL_runDF22_L3	34	
DCL_runDF22_L4	83	146
DCL_runDF23_C1	62	64
DCL_runDF23_C2	20	69
DCL_runDF23_C3	54	
DCL_runDF23_C4	98	107
DCL_runDF23_C5	29	26

DCL_runDF23_C6	82	97
DCL_runDF23_L1	44	60
DCL_runDF23_L2	20	80
DCL_runDF23_L3	44	
DCL_runGSM_C1	50	TBD
DCL_runNLF_C1	1075[1]	1075[1]
DCL_runPIDF64_S1	2330	310
DCL_writeLog	48	N/A
DCL_readLog	39	N/A
DCL_freadLog	22	11
DCL_fwriteLog	22	14
DCL_runClamp_C1	28	20
DCL_runClamp_C2	71	
DCL_runClamp_L1	25	26
DCL_runITAE_C1	[3]	60
DCL_runIAE_C1	[3]	
DCL_runIES_C1	[3]	

[1] All paths operating in linearized error region. For all paths in non-linear operation, total cycle count is approximately 1,433. See section 3.3.1 for more information.

[2] Measured with run-time library support for the `pow()` function.

[3] Cycle count depends on buffer length. Refer to section 3.4.1 for more information.

The table below shows execution benchmarks for the supporting functions new in v3.0 of the DCL. Execution figures include C function calling overhead. Separate measurements are given for update performed and not performed according to the `sts` flag status (see section 2.4). In all cases, compiler optimization was turned off.

Table 4. Support function execution benchmarks

Function	**File**	**Cycles [1]**	**Cycles [2]**
DCL_resetPID	DCLF32.h	85	85
DCL_updatePID	DCLF32.h	964 / 796	201 / 33
DCL_setPIDfilterBW	DCLF32.h	1060	771
DCL_getPIDfilterBW	DCLF32.h	509	509
DCL_loadSeriesPIDasZPK	DCLF32.h	3226	2939
DCL_loadParallelPIDasZPK	DCLF32.h	2741	2454
DCL_resetPI	DCLF32.h	71	71
DCL_updatePI	DCLF32.h	326 / 191	166 / 33
DCL_loadSeriesPIasZPK	DCLF32.h	668	306
DCL_loadParallelPIasZPK	DCLF32.h	433	71
DCL_resetPI2	DCLCLA.h	82	82

DCL_updatePI2	DCLCLA.h	267 / 179	131 / 33
DCL_resetDF11	DCLF32.h	71	71
DCL_updateDF11	DCLF32.h	113 / 33	113 / 33
DCL_isStableDF11	DCLF32.h	62 / 62	62 / 62
DCL_loadDF11asPI	DCLF32.h	611	548
DCL_loadDF11asZPK	DCLF32.h	291	102
DCL_resetDF13	DCLF32.h	110	110
DCL_updateDF13	DCLF32.h	200 / 33	201 / 33
DCL_isStableDF13	DCLF32.h	1532 / 1524	1532 / 1524
DCL_loadDF13asZPK	DCLF32.h	5333	4962
DCL_resetDF22	DCLF32.h	72	72
DCL_updateDF22	DCLF32.h	144 / 33	144 / 33
DCL_isStableDF22	DCLF32.h	794	794 / 793
DCL_loadDF22asZPK	DCLF32.h	1419	1155
DCL_loadDF22asParallelPID	DCLF32.h	1208	826
DCL_loadDF22asSeriesPID	DCLF32.h	1191	823
DCL_resetDF23	DCLF32.h	78	78
DCL_updateDF23	DCLF32.h	182 / 33	182 / 33
DCL_loadDF23asZPK	DCLF32.h	7138	4950
DCL_isStableDF23	DCLF32.h	1531 / 1516	1531 / 1516
DCL_resetPID32	DCL32.h	52	52
DCL_updatePID32	DCL32.h	154 / 34	154 / 34
DCL_resetPI32	DCL32.h	32	32
DCL_updatePI32	DCL32.h	103 / 34	103 / 34
DCL_resetGSM	DCL.h	327	327
DCL_updateGSM	DCL.h	504 / 33	504 / 33
DCL_loadGSMgains	DCL.h	389	389
DCL_loadGSMoffsets	DCL.h	501	501
DCL_runGSM_C1	DCL.h	73 / 52	73 / 52
DCL_fupdatePID	DCL_futils.asm	77 / 37	N/A
DCL_fupdatePI	DCL_futils.asm	64 / 37	N/A
DCL_fupdatePI2	DCL_futils.asm	57 / 37	N/A
DCL_fupdateDF11	DCL_futils.asm	58 / 37	N/A
DCL_fupdateDF13	DCL_futils.asm	78 / 37	N/A
DCL_fupdateDF22	DCL_futils.asm	66 / 37	N/A
DCL_fupdateDF23	DCL_futils.asm	74 / 37	N/A
DCL_fupdateGSM	DCL_futils.asm	114 / 37	N/A
DCL_fupdatePID32	DCL_futils32.asm	73 / 37	N/A
DCL_fupdatePI32	DCL_futils32.asm	57 / 37	N/A
DCL_resetNLPID	DCL_NLPID.h	84	85
DCL_updateNLPID	DCL_NLPID.h	1190 / 871	353 / 33

DCL_setNLPIDfilterBW	DCL_NLPID.h	1066	771
DCL_setActiveNLPIDfilterBW	DCL_NLPID.h	1028	743
DCL_getNLPIDfilterBW	DCL_NLPID.h	509	509
DCL_getNLPIDgamma	DCL_NLPID.h	998	999
DCL_getNLPIDdelta	DCL_NLPID.h	1273	1274
DCL_setNLPIDgamma	DCL_NLPID.h	3284	3057
DCL_getNLFgamma	DCL_NLPID.h	995	995
DCL_setActivePIDfilterBW	DCLF32.h	1027	743
DCL_resetPIDF64	DCLF64.h	106	106
DCL_updatePIDF64	DCLF64.h	546 / 35	237 / 33
DCL_setPIDF64filterBW	DCLF64.h	10599	8132
DCL_setActivePIDF64filterBW	DCLF64.h	10503	7982
DCL_getPIDF64filterBW	DCLF64.h	5163	5135

(1) Built with no optimization and error checking enabled.

(2) Built with no optimization and error checking disabled.

The table below shows execution benchmarks for the fast update functions of the DCL. All functions are implemented in assembly code. The right-most column shows the number of CPU clock cycles for which global interrupts are disabled while the copy takes place. Interrupts are not disabled if an update is not pending and no update is performed.

Table 5. Fast update function execution benchmarks

Function	File	Cycles (update)	Cycles (no update)	Interrupts blocked (cycles)
DCL_fupdatePID	DCL_futils.asm	77	37	37
DCL_fupdatePI	DCL_futils.asm	64	37	24
DCL_fupdatePI2	DCL_futils.asm	57	37	17
DCL_fupdateDF11	DCL_futils.asm	58	37	31
DCL_fupdateDF13	DCL_futils.asm	78	37	38
DCL_fupdateDF22	DCL_futils.asm	66	37	26
DCL_fupdateDF23	DCL_futils.asm	74	37	34
DCL_fupdateGSM	DCL_futils.asm	114	37	74
DCL_fupdatePID32	DCL_futils32.asm	73	37	33
DCL_fupdatePI32	DCL_futils32.asm	57	37	17

Chapter 2

Using the Digital Control Library

This chapter describes how to use the Digital Control Library.

Section

2.1 What the Library Contains

2.2 Header File Dependency

2.3 How to Add the DCL to User Code

2.4 Updating Controller Parameters

2.5 Error Handling

2.6 How to Modify the Library Code

2.1 What the Library Contains

The DCL library is supplied entirely in open source format. There are no object or ".lib" files in the library. This allows the user to modify the library controller functions, or to add their own functions if different functionality is required. Controller functions are supplied in the following formats.

- Inline C code
- FPU32 assembly code
- C28x assembly code
- CLA assembly code

2.1.1 Header Files

The following header files are included in the library.

Table 6. List of DCL header files

Filename	Type	Description
DCL	h	Common library definitions
DCLF32	h	FPU32 controller functions
DCLF64	h	Double precision controller functions

DCLCLA	h	CLA controller functions
DCLC28	h	C28x fixed point controller functions
DCL_NLPID	h	FPU32 non-linear PID
DCL_MLOG	h	Four channel FPU32 data logger
DCL_fdlog	h	FPU32 data logger
DCL_fdlog64	h	Double precision data logger
DCL_TCM	h	FPU32 Transient Capture Module
DCL_log32	h	C28x data logger functions
DCL_MLOG32	h	C28x four channel data logger

2.1.2 Source Files

The following source files are included in the library.

Table 7. List of DCL source files

Filename	Type	CPU	Description
DCL_PID_A1	asm	C28x	Fixed-point linear PID
DCL_PID_C1	asm	FPU32	Ideal linear PID
DCL_PID_C4	asm	FPU32	Parallel linear PID
DCL_PID_L1	asm	CLA	Ideal linear PID
DCL_PID_L2	asm	CLA	Parallel linear PID
DCL_PI_A1	asm	C28x	Fixed-point linear PI
DCL_PI_C1	asm	FPU32	Ideal linear PI
DCL_PI_C4	asm	FPU32	Parallel linear PI
DCL_PI_L1	asm	CLA	Ideal linear PI
DCL_PI_L2	asm	CLA	Parallel linear PI
DCL_DF11_C1	asm	FPU32	Full DF1 (1^{st} order)
DCL_DF11_L1	asm	CLA	Full DF1 (1^{st} order)
DCL_DF13_C1	asm	FPU32	Full DF1 (3rd order)
DCL_DF13_C2C3	asm	FPU32	Pre-computed DF1 (3rd order)
DCL_DF13_L1	asm	CLA	Full DF1 (3rd order)
DCL_DF13_L2L3	asm	CLA	Pre-computed DF1 (3rd order)
DCL_DF22_C1	asm	FPU32	Full DF2 (2nd order)
DCL_DF22_C2C3	asm	FPU32	Pre-computed DF2 (2nd order)
DCL_DF22_L1	asm	CLA	Full DF2 (2nd order)
DCL_DF22_L2L3	asm	CLA	Pre-computed DF2 (2nd order)
DCL_DF23_C1	asm	FPU32	Full DF2 (3rd order)
DCL_DF23_C2C3	asm	FPU32	Pre-computed DF2 (3rd order)
DCL_DF23_L1	asm	CLA	Full DF2 (3rd order)
DCL_DF23_L2L3	asm	CLA	Pre-computed DF2 (3rd order)
DCL_frwlog	asm	FPU32	Fast read/write log functions
DCL_clamp_C1	asm	FPU32	Data clamp
DCL_clamp_L1	asm	CLA	Data clamp
DCL_index	asm	FPU32	Performance measurement

DCL_error	C	FPU32	Template error handler
DCL_futils	asm	FPU32	Fast parameter updates
DCL_futils32	asm	C28x	Fast parameter updates

2.1.3 Examples

A set of code examples is supplied with the Digital Control Library. These were prepared using CCS version 7 and run without modification on either the F28069 or F280049 device. The examples include linker command files which show how to allocate device memory when using the DCL. Further details can be found in chapter 5.

Table 8. List of DCL code examples

Example	Description	Core	Device
1	DF22 compensator	FPU32	F28069
2	DF32 compensator	CLA	F28069
3	NLPID controller	FPU32	F28069
4	PI controller	CLA	F28069
5	PID controller	FPU32	F28069
6	TCM	FPU32	F28069
7	Smith predictor	FPU32	F28069
8	Gain scheduler	FPU32	F28069
9	Multiple control loops + ERAD	FPU32	F280049

2.2 Header File Dependency

A major change between v2.1.1 and v3.0 is header file dependency. From v3.0 onwards, the file `DCL.h` is a central repository of common data types and sub-structure definitions. FPU32 controller functions which were previously located in this file may now be found in the new header file `DCLF32.h`. Similarly, CLA controller functions are now located in the file `DCLCLA.h`. These changes provide a more logical separation of controllers based on their CPU type and make future expansion of the library more straightforward.

The convention in v3.x is that header files which contain basic controller functions for a specific CPU are named "`DCLxyz.h`", where "`xyz`" identifies the CPU designator and there is no underscore. Header files which contain specific controllers or utilities have an underscore immediately following the library designator, for example "`DCL_NLPID.h`". Header file dependency is shown diagrammatically below.

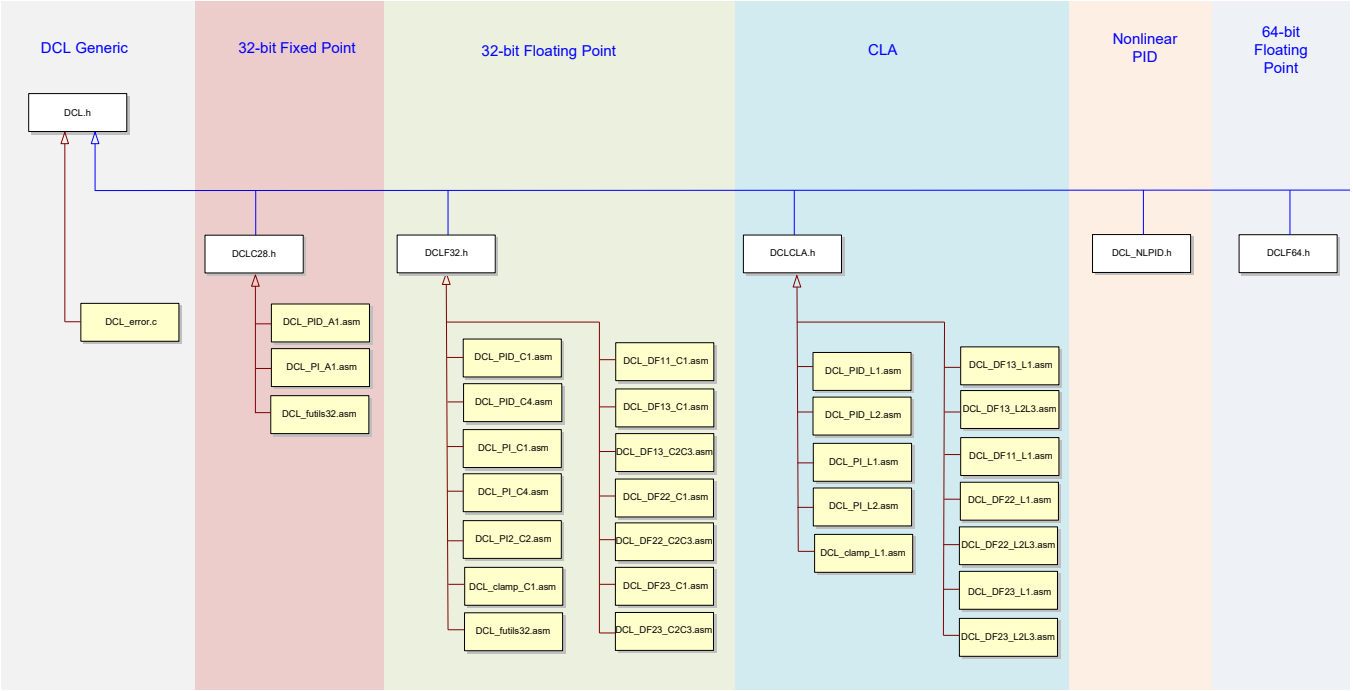


Figure 2. *DCL Header File Dependency*

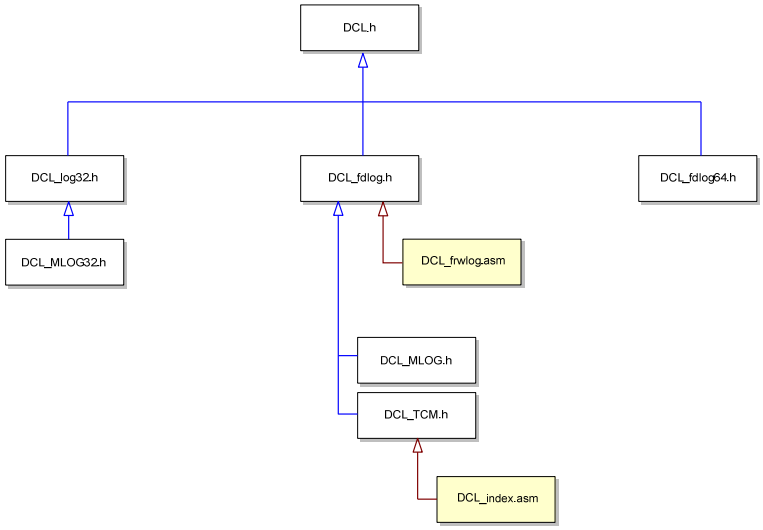


Figure 3. *DCL Data Logger Header Dependency*

2.3 How to Add the DCL to User Code

The Digital Control Library is intended to be used with a CCS project written in the C programming language. The user is responsible for initializing elements of the controller structure prior to calling the controller 'run' function. Typically the desired controller

functions would be inserted into an Interrupt Service Routine (ISR) triggered by a hardware event, which ensures that they are executed at a fixed rate and that their timing is synchronized with the availability of incoming data. Control functions for use on the CLA would be called from a CLA task, which again, would typically be triggered at a fixed rate by a hardware event.

Controller parameters and dynamic data are contained in a C structure, a pointer to which is passed as a parameter to the controller 'run' function. Typically, the controller structure would be a global variable in the user's program and its' contents initialized prior to the first call to the 'run' function. The controller 'run' functions are not re-entrant, since they rely on a global variable (in this case a structure). It is the responsibility of the user to ensure that a controller function is not called while a similar controller function is in progress.

2.3.1 Steps to Add the DCL to Existing C Code

The following is a recommended sequence of steps which can be followed when adding the DCL to an existing C program. Refer to chapter 5 for code examples which illustrate configuration and use of the DCL with the CLA.

Step 1. Specify the include file(s)

Before you can begin using the library you must add the appropriate controller header file to your project. To use the 32-bit floating-point DCL functions include the file `DCLF32.h`.

```
#include    "DCLF32.h"
```

To use fixed-point CLA functions include the file `DCLCLA.h`.

```
#include    "DCLCLA.h"
```

To use fixed-point DCL functions include the file `DCLC28.h`.

```
#include    "DCLC28.h"
```

To use double precision floating point DCL functions include the file `DCLF64.h`.

```
#include    "DCLF64.h"
```

It is not necessary to include the common library file `DCL.h`.

CCS must be configured in such a way that the DCL header files are visible to all program source files which reference controller variables or functions. The include file search options in CCS allow users to specify header file paths for each project. In CCS, the include options can be configured by right-clicking on the project name, selecting "Properties", and navigating to the "Include Options" section.

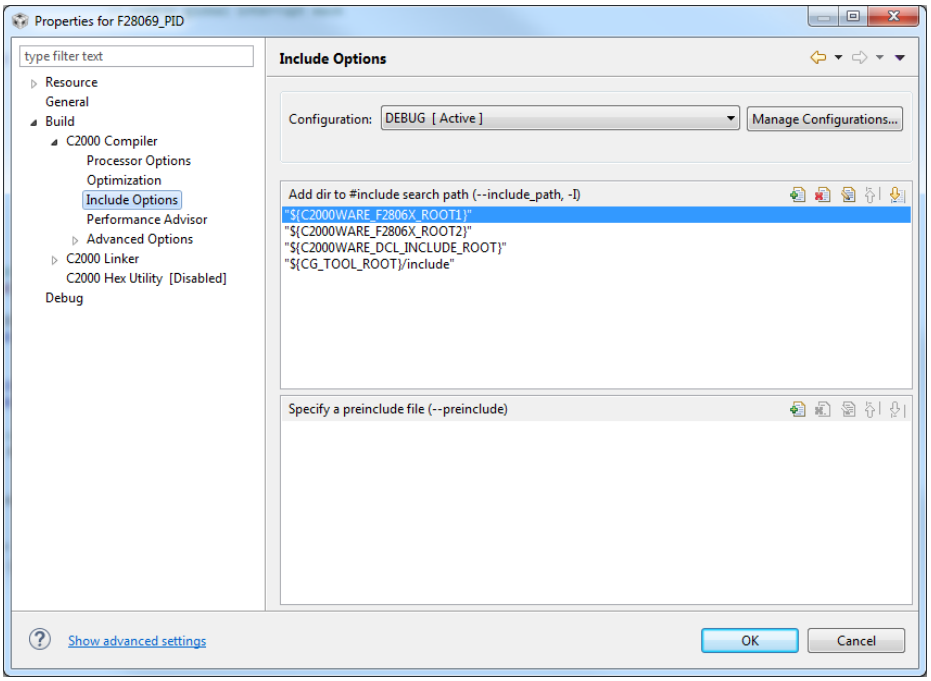


Figure 4. *CCSv6 Include Options*

If you wish to use the data logger, MLOG, or TCM, you must also include the respective header file(s) (note that the MLOG and TCM modules include the relevant data log header file). If you wish to use the non-linear controller functions, the header file `DCL_NLPID.h` must be included in your program. Refer to section 2.2 for more information on header file dependency.

Step 2. Add the source files to the project

If you wish to use any of the assembly coded controllers, the source file(s) for the controller(s) you wish to use must be added to your CCS project. You can manually copy the files into your project directory, or specify the library pathname in the CCS compiler options. Refer to Table 7 for a list of controller source files. It is only necessary to add the source files for those functions you wish to use.

Step 3. Allocate the controller functions in the linker command file

DCL functions which run on the FPU32 or C28x core can be allocated to a specific memory block in the linker command file. It is common to place the controller functions in zero wait-state internal RAM, since this allows them to run at the maximum speed of the device. Note that all CLA functions must run from internal zero wait state RAM.

All DCL library functions are placed in the user defined code section `.dclfuncs`. An example showing how this section might be mapped into the internal L4 RAM memory block is shown below.

```
dclfuncs            : > RAML4,          PAGE = 0
```

See also the linker command file `F28069_DCL.cmd` in the project examples.

In a stand-alone application, code must be stored in non-volatile memory (such as internal flash) and copied into RAM at run-time. For information on how to do this, refer to the application note "*Running an Application from Internal Flash Memory on the TMS320F28xxx DSP*", TI literature number SPRA958.

Information on linker section allocation can be found in the "*TMS320C28x Assembly Language Tools User's Guide*" (see section 6.1).

Step 4. Create an instance of the controller

You must declare an instance of the controller you wish to use. For example, to create an initialized instance of a PID controller with the name "pid1":

```
DCL_PID pid1 = PID_DEFAULTS;
```

This step will create an instance of a PID controller the elements of which are loaded with default values specified in the file `DCLF32.h`.

Note that CLA variables must be initialized at run-time by user code (i.e. they cannot be initialized at the variable declaration). Typically this is done using a separate CLA task (see code examples 2 & 4).

Step 5. Create instances of the support structures (optional)

If you wish to make use of the error checking or safe parameter update features in the library, you must declare instances of both the SPS and CSS sub-structures, and initialize them appropriately. If you do not wish to use these features, this step can be ignored.

To declare an initialized instance of the SPS for the PID controller, you would do the following:

```
DCL_PID_SPS spid = PID_SPS_DEFAULTS;
```

To declare an initialized instance of the CSS for the PID controller, you would do the following:

```
DCL_CSS cpid = DCL_CSS_DEFAULTS;
```

Assign each of the above structures to the PID control structure in step 4.

```
pid1.sps = &spid;

pid1.css = &cpid;
```

This creates a variable of type "DCL_PID", the elements of which are initialized to default values specified in the `DCL.h` header file. Like any C variable, the structure must be visible to any source files which reference it.

Step 6. Declare variables

In addition to a pointer to the controller structure, each DCL controller function requires certain input variables to be passed as arguments to the function, and will return a control output. You should declare instances of these variables in your code and ensure they can be referenced by all files which call the controller functions. For example:

```
float uk;          // control

float rk = 0.0f;   // reference

float yk = 0.0f;   // feedback

float lk = 1.0f;   // saturation
```

Note that CLA variables cannot be initialized at the declaration. Refer to the CLA code in chapter 5 for examples of how to initialize CLA variables.

Step 7. Initialize the controller

The elements of the (FPU32 or C28x) controller structure were initialized to default settings in step 3. The user program must configure any controller elements with specific values before the function is called. For example:

```
pid1.Kp = 9.4f;    // set proportional gain to 9.4
pid1.Umax = 10.0f; // upper output clamp limit = 10
```

If a CLA based controller is being used, its parameters must always be initialized using a separate task. For more information on the CLA C compiler, see Chapter 10 of the "*TMS320C28x Optimizing C/C++ Compiler User's Guide*" (see section 6.1).

Direct Form control structures incorporate one or two delay lines which hold previous controller data. These must be initialized to zero before calling the controller functions, which can be done with the appropriate "DCL_reset" function. If this is not done, it is possible that uninitialized delay line data, especially in the recursive path, might cause the controller to saturate or deliver incorrect results. Initialization of the delay line elements is the responsibility of the user. Refer to the code in examples 1 and 2 described in chapter 4 for examples of delay line initialization.

Step 8. Call the controller function

Typically the controller functions would be inserted into an Interrupt Service Routine (ISR) which is triggered by a hardware timer. This ensures that the control law is executed at a deterministic and fixed time interval. Each control function returns a single floating-point variable which represents the controller output. An example of a controller function call is shown below.

```
uk = DCL_runPID(&pid1, rk, yk  lk);
```

2.3.2 Calling the Library Functions from Assembly

The assembly coded functions in the DCL have been written to be called from a C program. The context save and restore sections within each function protect only those core registers which are not already protected by the C environment. In applications where the DCL controller functions must be called from an assembly program, the user must place additional register save and restore instructions near the start end end of each called function.

For the C28x, refer to sections 7.2 and 7.3 of the "*TMS320C28x Optimizing C/C++ Compiler User's Guide*" (see section 6.1) for detailed information on register usage and calling conventions. For the CLA, refer to section 10.2.4 of the same document.

2.4 Updating Controller Parameters

Version 3.x of the DCL allows controller parameters to be safely updated "on-the-fly", without stopping the control loop. To accomplish this, the user must declare and initialize the SPS and CSS sub-structures as described in section 2.3.1, step 5.

To update controller parameters, follow these steps:

Step 1. Load the shadow parameter set

Load all the elements of the SPS sub-structure with the new controller values. Continuing the example in section 2.3.1, one might do this.

```
pid1.sps.Kp = 10.987f;
pid1.sps.Ki = 0.0023f;
```

Ensure all the SPS elements are loaded before performing the update, even if they do not need to change.

Step 2. Enable the update flag

Set the update flag in the CSS status register to enable the parameter copy on the next call to the update function. There is a C macro in `DCL.h` to do this.

```
DCL_REQUEST_UPDATE(&pid1);
```

This sets the LSB in the status element `sts` in the CSS sub-structure.

Step 3. Call the update function

Call the appropriate controller update function. This would typically be done in the background loop.

```
DCL_updatePID(&pid1);
```

This function tests the update flag to determine whether an update is pending. If so, it blocks the device interrupts, performs the shadow to active parameter set copy, re-enables device interrupts, and clears the update request flag previously set in step 1.

In blocking interrupts, the library will save the prior state of the global interrupt flag (INTM) into a local variable. The state is restored when interrupts are re-enabled at the end of the update. In this way, update functions may be called by the user irrespective of whether interrupts are already enabled: the function will not change the prior interrupt state. A similar method of interrupt blocking and protection is implemented in the 'reset' functions to ensure controllers do not run with partially reset internal variables. Interrupt blocking macros and functions can be found in the library header file `DCL.h`.

The updating of controller parameters in the above fashion is a time critical task, since interrupts must be disabled while copying takes place. For this reason the library includes a set of assembly coded update functions which are used in the same way described above, but which execute faster than their C counterparts and are deterministic. In all cases, the same update function names are pre-fixed with the letter 'f', so the above example becomes

```
DCL_fupdatePID(&pid1);
```

The floating point assembly update routines are contained in the source file `DCL_futils.asm` which must be added to the user's project. At the top of this file is a list of "`.set`" directives which allows the user to selectively disable those functions which are not required in order to reduce code space.

Fixed point update routines are contained in the file `DCL_futils32.asm`.

The table below lists the fast parameter update functions together with the cycle counts when an update is performed and when the update is by-passed. Also shown are the number of CPU cycles for which global interrupts are masked when the update takes place. Note that the assembly functions do not perform any error checking prior to the update.

Table 9. List of fast parameter update function execution cycles

Function	Update taken	No update	Interrupts blocked
DCL_fupdatePID	77	37	37

DCL_fupdatePI	64	37	24
DCL_fupdatePI2	57	37	17
DCL_fupdateDF11	58	37	31
DCL_fupdateDF13	78	37	38
DCL_fupdateDF22	66	7	26
DCL_fupdateDF23	74	37	34
DCL_fupdateGSM	114	37	74
DCL_fupdatePID32	73	37	33
DCL_fupdatePI32	57	37	17

2.5 Error Handling

The DCL contains limited support for error detection and handling. Many of the supporting functions perform range checks on parameters and input variables to ensure they fall within allowable ranges. These checks consume cycles, and in performance critical situations the user may elect to disable error checking by commenting out thte following line in `DCL.h`.

```
#define DCL_ERROR_HANDLING_ENABLED
```

Users should inspect the source code for the relevant functions to determine which checks are performed.

If an error is detected, the code will set the "`err`" field of the CSS sub-structure of the controller with an error code. An enumerated list of error codes can be found in `DCL.h`.

Table 10. List of CSS enumerated error codes

Name	Description
ERR_NONE	No error present
ERR_PARAM_RANGE	A parameter was passed to a function, or a controller element found, which lay outside its allowable range.
ERR_PARAM_INVALID	An invalid parameter was passed to a DCL function.
ERR_PARAM_WARN	A non-critical parameter error was found.
ERR_INPUT_RANGE	An input was supplied which lies outside the allowable range.
ERR_OVERFLOW	A variable exceeded its' allowable range.
ERR_UNDERFLOW	A variable was below its' allowable range.
ERR_VERSION	The version of the DCL is incorrect

ERR_DEVICE	A function was called which requires hardware features not present on the selected device.
ERR_CONTROLLER	A control operation was called before the same operation had completed.

After each test sequence, if the `err` field is non-zero, the code will load the line number of the error and call an error handler function using the following C macros.

```
if (p->css->err)
 {
    DCL_GET_ERROR_LOC(p->css);
    DCL_RUN_ERROR_HANDLER(p->css);
 }
```

The default error handler function is located in the source file `DCL_error.c`. The DCL does not perform sophisticated error handling, however the user is free to add their own code to this file, or to re-direct the error handler to their own custom error function if desired. Note that the enumerated error list is likely to be appended in future versions of the library.

Among those errors in the standard enumerated list is ERR_CONTROLLER. This error is used to detect over-run conditions, where a non-atomic controller function fails to complete before being called again. This is potentially dangerous since the second call will run with a partially updated data set. To detect this condition, the file `DCL.h` defines the macros DCL_CONTROLLER_BEGIN and DCL_CONTROLLER_END which the user may place at the start and end of a controller function to set and clear respectively the STS_CONTROLLER_RUNNING bit in the STS field of the CSS sub-structure. This can be examined inside the function to detect a partially complete control operation as follows.

At the start of the controller function: check whether the STS_CONTROLLER_RUNNING bit is set, then set it.

```
p->css->err |= (p->css->sts & STS_CONTROLLER_RUNNING) ? ERR_CONTROLLER :
ERR_NONE;
if (p->css->err)
{
    DCL_GET_ERROR_LOC(p->css);
    DCL_RUN_ERROR_HANDLER(p->css);
}
DCL_CONTROLLER_BEGIN(p);
```

At the end of the controller function: clear the STS_CONTROLLER_RUNNING bit.

```
DCL_CONTROLLER_END(p);
```

In the above code, `p` represents a pointer to the controller structure. Refer to the controller code in `DCL_NLPID.h` for examples of the above method.

2.6 How to Modify the Library Code

The DCL is supplied entirely in source code form so it is possible (indeed, encouraged) for users to modify the functions freely to meet specific needs or improve performance.

However, the library is in continuous development at TI and users should exercise care to ensure any changes do not conflict with future releases. To modify the library while maintaining compatibility with future releases, the following precautions should be observed.

- Do not modify the DCL functions directly. To modify library code, first copy the code into a new function with a different name, then modify the new function. This will ensure that user code remains compatible with future library releases.
- Select a name which will not conflict with future library versions. See section 1.3.2 for a description of function naming. Controller numbers of 20 and above are reserved for customer use. For example, the TI library will never contain a function `DCL_runPI_C20`, so users are free to use that function name for their own code.
- Users are not obliged use the DCL controller structures, however they are free to do so. It is suggested that any new custom structures use a name which is pre-fixed differently from the library, for example, by replacing DCL_ with DCLU_.

```
typedef volatile struct {
      float32_t Kp;
      float32_t Ki;
      …
      } DCLU_PID;
```

- Users need not apply the parameter update or error checking methods described in sections 2.4 and 2.5, however they are free to do so. It is intended that future library versions will be compatible with v3.x in this respect.

Chapter 3

Controllers

This chapter provides detailed information on the controller functions in the Digital Control Library.

Section

3.1 Linear PID Controllers

3.2 Linear PI Controllers

3.3 Non-linear PID Controllers

3.4 Non-linear PI Controller

3.5 Double Integrator PI Controller

3.6 Direct Form 1 (first order) Compensators

3.7 Direct Form 1 (third order) Compensators

3.8 Direct Form 2 (second order) Compensators

3.9 Direct Form 2 (third order) Compensators

3.10 Fixed Point PID Controllers

3.11 Fixed Point PI Controllers

3.12 Gain Scheduler Module

3.13 Non-linear Law

3.14 Double Precision PID Controllers

The DCL contains ten basic types of controller.

- Linear PID
- Linear PI
- Non-linear PID
- Non-linear PI
- Double Integrator
- Direct Form 1 (first order)
- Direct Form 1 (third order)

- Direct Form 2 (second order)
- Direct Form 2 (third order)
- Gain scheduler

In this guide, the four direct form types are referred to as ‘compensators’. This reflects a situation typical in power supply design, in which the objectives are to compensate some feature of the open loop frequency response, such as phase shift. The compensator is usually specified using a set of pole & zero frequencies, which leads naturally to a transfer function description. The term “Direct Form” comes from transfer function descriptions of digital filters.

Controllers are either coded in C, or in assembly using three different instruction sets (C28x, FPU32, and CLA). Additionally, most of the Direct Form compensators are implemented in both full and pre-computed forms. There may therefore be several different functions for each controller to allow the user to balance execution time with ease-of-tuning. Different implementation of the same controller are identified using a two character suffix to the ‘run’ function name (refer to section 1.3.2 for information on function naming).

The non-linear PID controllers, 64-bit floating point PID, and the double integrator PI are currently only available in C code form. At the present time, support for the `pow()` function used in the non-linear control law is only available in the standard C run-time support library. Note that this controller is not supported on the CLA.

The description of each controller in this chapter is broken down into three sub-sections:

- A general description of the controller
- Information of the implementation of the controller
- A description of functions

The ‘implementation’ sub-section always includes a block diagram showing the internal structure and the variables used in the code. Local variables, which do not need to be preserved between functions, are pre-fixed with the letter “v”. Variables which are part of the controller structure and are therefore preserved between function calls are pre-fixed with some other letter according to their purpose: for example, “i10” refers to a variable used in the PID integrator. The prefix letter “l” is reserved for logical signals. The same variable names are used in the library source code, making it straightforward to compare the controller diagrams with the source code.

It is sometimes useful to monitor internal controller variables (i.e. which are not elements in the controller structure) for debugging purposes. The CSS sub-structure contains a test-point element named “`tpt`” which is intended to be used for this purpose. Controller test-points are globally enabled using the following definition in `DCL.h`. Note that by default, this definition is commented out to reduce execution cycles.

```
#define DCL_TESTPOINTS_ENABLED
```

Users may un-comment the above line, and then assign any internal variable to the `tpt` element for monitoring, perhaps in the CCS “Expressions” window or using a data logger.

3.1 Linear PID Controllers

3.1.1 Description

The basic controller type described here is a linear PID. The PID implementations in the DCL include several features not commonly found in basic PID designs, and this complexity is reflected the benchmark figures. Applications which do not require derivative action, or are more sensitive to cycle efficiency, may be better served by the simpler PI controller structure described in the next section.

PID control is widely used in systems which employ output feedback control. In such systems, the controlled output is measured and fed back to a summing point where it is subtracted from the reference input. The difference between the reference and feedback corresponds to the control loop error and forms the input to the PID controller.

Conceptually, the PID controller output is the parallel sum of three paths which act respectively on the error, error integral, and error derivative. The relative weight of each path may be adjusted by the user to optimize transient response, or to emulate the behavior of a specified transfer function expressed in terms of its' poles and zeros.

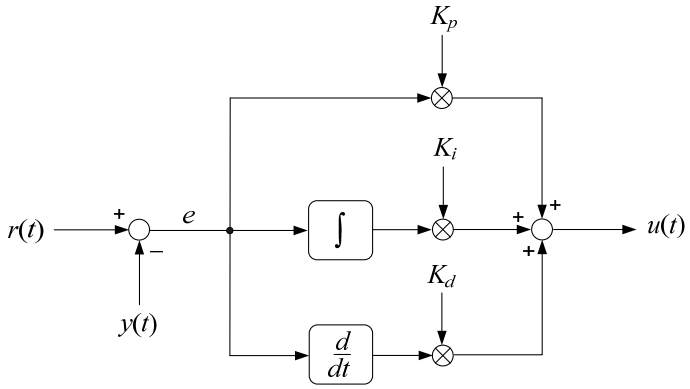


Figure 5. Parallel form PID controller

The diagram above shows the structure of a continuous time 'parallel' PID controller. The output of this controller is captured in the following equation:

Equation 2. $$u(t) = K_p\, e(t) + K_i \int_{-\infty}^{t} e(\tau)\, d\tau + K_d \frac{de(t)}{dt}$$

Conceptually, the controller comprises three separate paths connected in parallel. The upper path contains an adjustable gain term (Kp). Its effect is to fix the open loop gain of the control system. Since loop gain is proportional to this term, Kp is known as proportional gain.

A second path contains an integrator which accumulates error history. A separate gain term acts on this path. The output of the integral path changes continuously as long as a non-zero error (e) is present at the controller input. A small but persistent servo error has the effect of driving the output of the integrator such that the loop error will eventually disappear. The principal effect of the integral path is therefore to eliminate steady state error. The effect of the integral gain term is to change the rate at which this happens.

Integral action is especially important in applications such as electronic power supplies, which must maintain accurate regulation over long periods of time.

The third path contains a differentiator. The output of this path is large whenever the rate of change of the error is large. The principal effects of the derivative action are to damp oscillation and reduce transients.

The operation of the PID controller can be visualized in terms of the transient error following a step change of set-point.

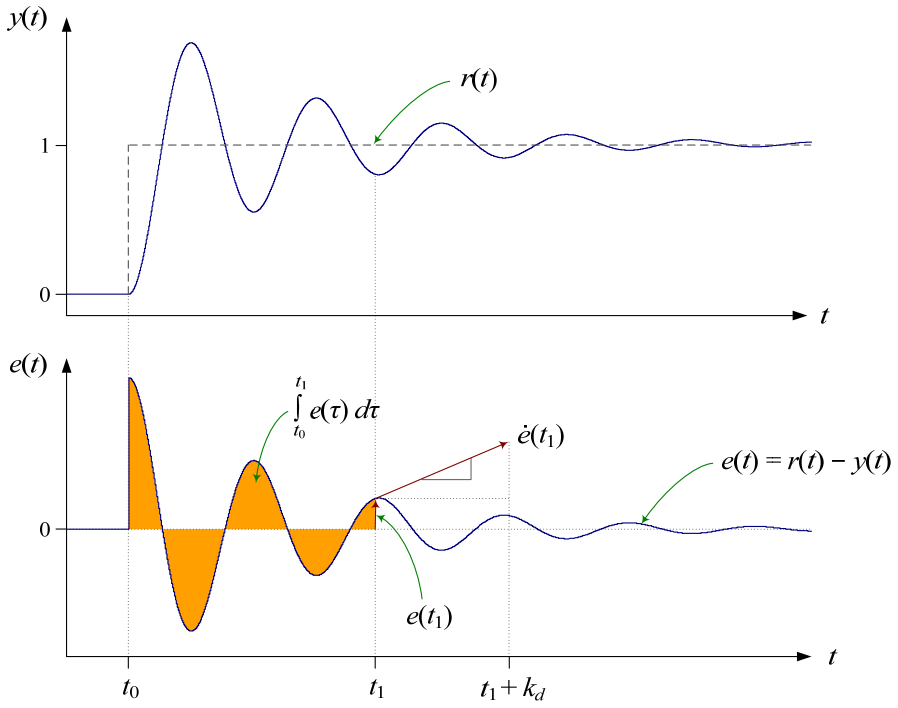


Figure 6. *PID control action*

The figure above shows the action of the PID controller in terms of the control loop error at time t_1. The proportional term contributes a control effort which is proportional to the instantaneous loop error. The output of the integral path is the accumulated error history: the shaded area in the lower plot. The contribution of the derivative path is proportional to the rate of change of the loop error. Derivative gain fixes the time interval over which a tangential line to the error curve is projected forward in time.

Tuning the PID controller is a matter of finding the optimum combination of these three effects. This in turn means finding the best balance of the three gain terms. For more information on PID control & tuning, see the references in section 6.1.

The PID shown above is known as the “parallel” form because the three controller gains appear in separate parallel paths. A slightly different PID architecture in which the proportional gain is moved into the output path (i.e. after the summing point), so that the proportional path becomes a direct connection between the controller input and the summing point, is known as the “series” or “ideal” form. In the ideal form, the open loop gain is directly influenced by the proportional controller gain, and there is less interaction between the controller gains. However the proportional gain cannot be zero (since the loop would be opened), and to maintain good control cannot be small. The parallel form

allows the proportional gain to be small, however there is slightly more interaction between the controller gains, complicating the tuning process. The DCL contains both ideal and parallel PID functions.

3.1.2 Implementation

The linear PID controllers in the DCL include the following features:

- Parallel and ideal forms
- Programmable output saturation
- Independent reference weighting on proportional path
- Anti-windup integrator reset
- Programmable low-pass derivative filter
- External saturation input for integrator anti-windup
- Adjustable output saturation

It is important to note that the controller sample period is not accounted for in the selection of integral gain (`Ki`). This is relevant when computing the integral gain as opposed to manual tuning against a transient response, for example. In such situations, users must multiply the computed integral gain by the sample period before loading `Ki` (either directly or through the SPS). The element `T` in the CSS sub-structure can be used to store the sample period for this purpose.

All PID type controllers in the library implement integrator anti-windup reset in a similar way. A clamp is present at the controller output which allows the user to set upper and lower limits on the control effort. If either limit is exceeded, an internal floating-point controller variable changes from 1.0 to 0.0. This variable is multiplied by the integrator input, such that the integrator accumulates zero data only when the output is saturated, thus preventing the well-known “wind-up” phenomenon.

The PID controllers in the library make provision for anti-windup reset to be triggered from an external part of the loop. This is useful in situations where a component outside the controller may be saturated. The floating-point variable “`lk`” is expected to be either 1.0 or 0.0 in the normal and saturated conditions respectively. If this feature is not required, the functions should be called with the `lk` argument set to 1.0. Note that all the controllers here require non-zero proportional gain to recover from loop saturation.

The derivative PID path includes a digital low-pass filter to avoid amplification of un-wanted high frequency noise. The filter implemented here is a simple first order lag filter (with differentiator), converted into discrete form using the Tustin transform. Referring to Figure 6, the difference equation of the filtered differentiator is

Equation 3. $$v_4(k) = v_1(k) - d_2(k) - d_3(k)$$

The temporary storage elements d_2 & d_3 must be preserved from the (k - 1)th interval, so the following must be computed after the differentiator update.

Equation 4. $$d_2(k) = v_1(k-1)$$

Equation 5. $d_3(k) = c_2 v_4(k-1)$

The derivative filter coefficients are

Equation 6. $c_1 = \dfrac{2}{T + 2\tau}$

Equation 7. $c_2 = \dfrac{T - 2\tau}{T + 2\tau}$

Both the sample period (T) and filter time constant (τ) must be determined by the user. The time constant is the reciprocal of the desired filter bandwidth in radians per second.

All linear PID controller functions use a common C structure to hold coefficients and data. Refer to the header file `DCLF32.h` for details of the `DCL_PID` controller structure.

The library PID controller architectures are shown below:

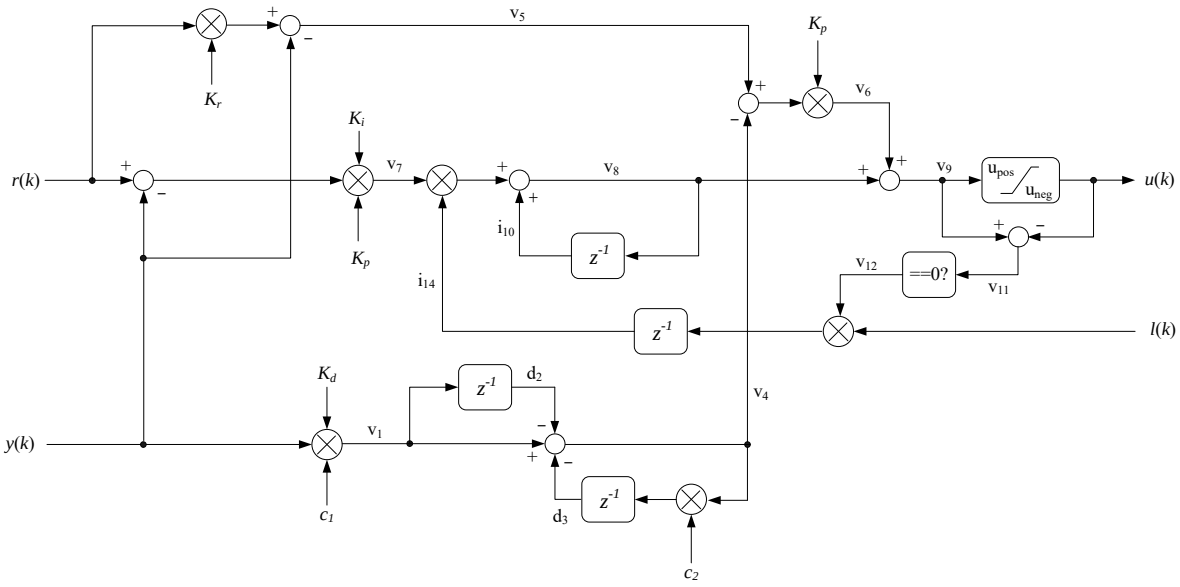


Figure 7. *DCL_PID C1, C2, & L1 architecture*

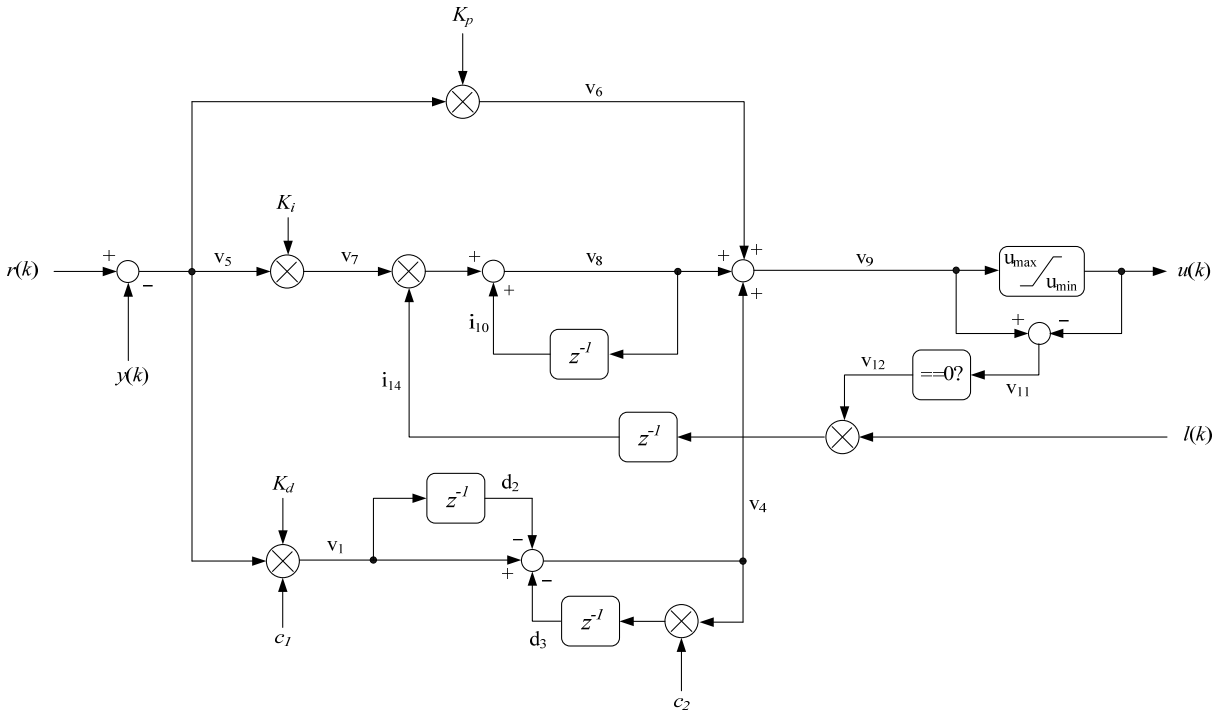


Figure 8. DCL_PID C3, C4, & L2 architecture

3.1.3 Functions

DCL_runPID_C1	***Run the Ideal Form PID Controller***

Header File: DCLF32.h

Source File: DCL_PID_C1.asm

Declaration: float32_t DCL_runPID_C1(DCL_PID *p, float32_t rk, float32_t yk, float32_t lk)

Description: This function executes an ideal form PID controller on the FPU32. The function is coded in assembly.

Parameters:

p	The DCL_PID structure
rk	The controller set-point reference
yk	The measured feedback
lk	External output clamp flag

Return: The control effort

DCL_runPID_C2	***Run the Ideal PID Form Controller***

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runPID_C2(DCL_PID *p, float32_t rk, float32_t yk, float32_t lk)

Description: This function executes an ideal form PID controller on the FPU32, and is identical in structure and operation to the C1 form. The function is coded in inline C.

Parameters:

p	The DCL_PID structure
rk	The controller set-point reference
yk	The measured feedback
lk	External output clamp flag

Return: The control effort

DCL_runPID_C3 — ***Run the Parallel Form PID Controller***

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runPID_C3(DCL_PID *p, float32_t rk, float32_t yk, float32_t lk)

Description: This function executes a parallel form PID controller on the FPU32. The function is coded in inline C.

Parameters:

p	The DCL_PID structure
rk	The controller set-point reference
yk	The measured feedback
lk	External output clamp flag

Return: The control effort

DCL_runPID_C4 — ***Run the Parallel Form PID Controller***

Header File: DCLF32.h

Source File: DCL_PID_C4.asm

Declaration: float32_t DCL_runPID_C4(DCL_PID *p, float32_t rk, float32_t yk, float32_t lk)

Description: This function executes a parallel form PID controller on the FPU32, and is identical in structure and operation to the C3 form. The function is coded in inline C.

Parameters:

p	The DCL_PID structure
rk	The controller set-point reference
yk	The measured feedback
lk	External output clamp flag

Return: The control effort

DCL_runPID_L1 — Run the Ideal Form PID Controller

Header File: DCLCLA.h

Source File: DCL_PID_L1.asm

Declaration: float32_t DCL_runPID_L1(DCL_PID_CLA *p, float32_t rk, float32_t yk, float32_t lk)

Description: This function executes an ideal form PID controller on the CLA. The function is coded in CLA assembly.

Parameters:

p	The DCL_PID_CLA structure
rk	The controller set-point reference
yk	The measured feedback
lk	External output clamp flag

Return: The control effort

DCL_runPID_L2 — Run the Parallel Form PID Controller

Header File: DCLCLA.h

Source File: DCL_PID_L2.asm

Declaration: float32_t DCL_runPID_L2(DCL_PID_CLA *p, float32_t rk, float32_t yk, float32_t lk)

Description: This function executes a parallel form PID controller on the CLA. The function is coded in CLA assembly.

Parameters:

p	The DCL_PID_CLA structure
rk	The controller set-point reference
yk	The measured feedback
lk	External output clamp flag

Return: The control effort

DCL_resetPID — Resets the PID Controller

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_resetPID(DCL_PID *p)

Description: This function resets the internal variables in the DCL_PID structure to default values. The integrator accumulator and store derivative path values are set to 0.0, and the integrator clamp variable set to 1.0. The function also sets the `err` field in the CSS sub-structure to NONE. Note that the function is atomic.

Parameters: p The DCL_PID structure

Return: Void

DCL_updatePID — ***Updates the PID Controller Parameters***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_updatePID(DCL_PID *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PID structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_PID structure

Return: Void

DCL_fupdatePID — ***Updates the PID Controller Parameters***

Header File: DCLF32.h

Source File: DCL_futils.asm

Declaration: void DCL_fupdatePID(DCL_PID *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PID structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters. The function is implemented as an assembly module.

Parameters: p The DCL_PID structure

Return: Void

DCL_setPIDfilterBW — ***Set the PID Derivative Filter Bandwidth***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_setPIDfilterBW(DCL_PID *p, float32_t fc)

Description: Loads the derivative filter coefficients `c1` & `c2` in the SPS based on the desired filter bandwidth specified in Hz. Coefficients in the active parameter set are unaffected until the controller is updated using DCL_updatePID().

Parameters:

p	The DCL_PID structure
fc	The desired filter bandwidth in Hz

Return: Void

DCL_setActivePIDfilterBW — ***Set the Active PID Derivative Filter Bandwidth***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_setActivePIDfilterBW(DCL_PID *p, float32_t fc, float32_t T)

Description: Loads the derivative filter coefficients `c1` & `c2` in the active PID structure based on the desired filter bandwidth specified in Hz and the controller update date in seconds. This function does not use or modify the SPS.

Parameters:

p	The DCL_PID structure
fc	The desired filter bandwidth in Hz
T	The controller update rate in seconds

Return: Void

DCL_getPIDfilterBW — ***Get the PID Derivative Filter Bandwidth***

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_getPIDfilterBW(DCL_PID *p)

Description: Finds the bandwidth of the current derivative filter in Hz by examining the coefficients in the active parameter set (i.e. not the SPS).

Parameters:

p	The DCL_PID structure

Return: The active derivative filter bandwidth in Hz

DCL_loadSeriesPIDasZPK — ***Load the Series Form PID Controller from ZPK***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadSeriesPIDasZPK(DCL_PID *p, DCL_ZPK3 *q)

Description: Loads the SPS coefficients to implement a series form PID controller (e.g. C1, C2 or L1) based on a ZPK3 definition. The ZPK3 is expected to be in the form of a complex zero pair, plus one real pole, plus an integrator. The real pole corresponds to the pole in the derivative path filter. Note that the active coefficients are not affected until the controller is updates using `DCL_updatePID()`. Refer to section 1.3.6 for more information on the ZPK3 structure.

Parameters: p The DCL_PID structure

q The ZPK3 structure

Return: Void

DCL_loadParallelPIDasZPK	***Load the Parallel Form PID Controller from ZPK***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadSeriesPIDasZPK(DCL_PID *p, DCL_ZPK3 *q)

Description: Loads the SPS coefficients to implement a parallel form PID controller (e.g. C3, C4 or L2) based on a ZPK3 definition. The ZPK3 is expected to be in the form of a complex zero pair, plus one real pole, plus an integrator. The real pole corresponds to the pole in the derivative path filter. Note that the active coefficients are not affected until the controller is updates using `DCL_updatePID()`. Refer to section 1.3.6 for more information on the ZPK3 structure.

Parameters: p The DCL_PID structure

q The ZPK3 structure

Return: Void

3.2 Linear PI Controllers

3.2.1 Description

The continuous time parallel PI control equation is

Equation 8. $$u(t) = K_p\, e(t) + K_i \int_{-\infty}^{t} e(\tau)\, d\tau$$

The linear PI controllers in the DCL differ from the PID in the following respects.

- Removal of derivative path
- Removal of set-point weighting

- No provision for external saturation input

In all other respects, the PI controllers are similar to the PID controllers described in section 3.1.

3.2.2 Implementation

All linear PI controller functions use a common C structure to hold coefficients and data, defined in the header files `DCLF32.h` and `DCLCLA.h`.

The PI controller architectures are shown in the following diagrams.

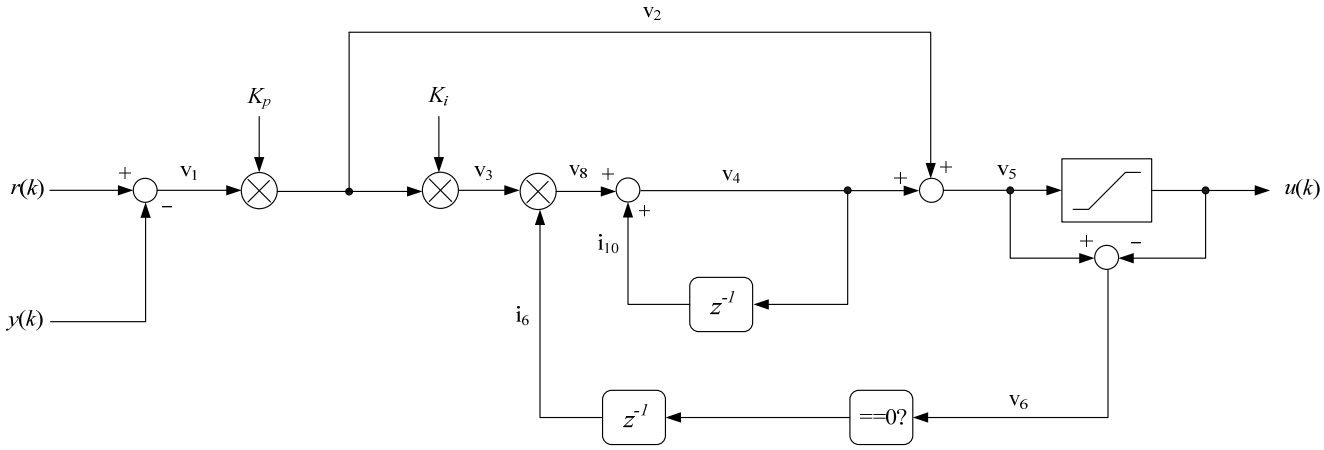


Figure 9. DCL_PI C1, C2, L1, & L3 architecture

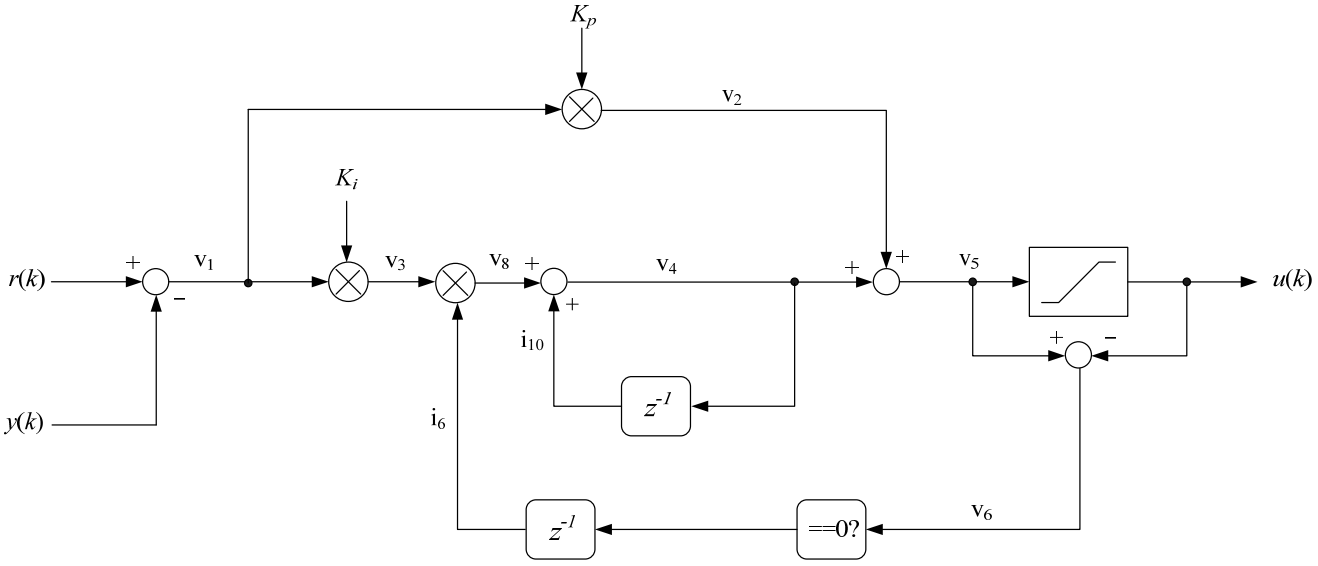


Figure 10. DCL_PI C3, C4, L2 & L4 architecture

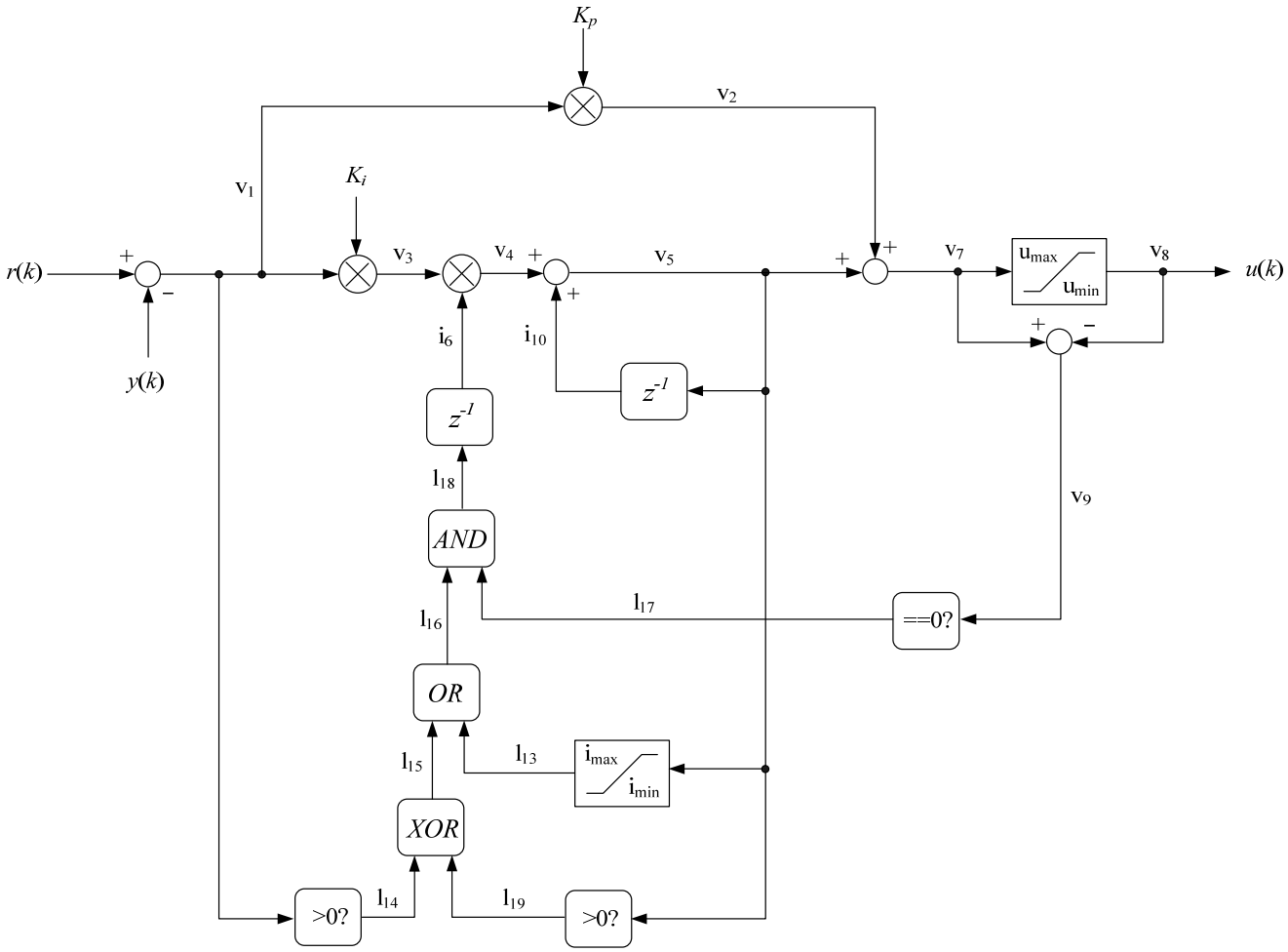


Figure 11. DCL_PI C5 architecture

Note that the C5 parallel form PI controller contains enhanced anti-windup reset logic which allows the integral path to recover from saturation even when the proportional gain is zero.

3.2.3 Functions

DCL_runPI_C1	**Run the Ideal Form PI Controller**

Header File: DCLF32.h

Source File: DCL_PI_C1.asm

Declaration: float32_t DCL_runPI_C1(DCL_PI *p, float32_t rk, float32_t yk)

Description: This function executes an ideal form PI controller on the FPU32. The function is coded in assembly.

Parameters:

p	The DCL_PI structure
rk	The controller set-point reference

	yk	The measured feedback
Return:	The control effort	

DCL_runPI_C2 — Run the Ideal PI Form Controller

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runPI_C2(DCL_PI *p, float32_t rk, float32_t yk)

Description: This function executes an ideal form PI controller on the FPU32, and is identical in structure and operation to the C1 form. The function is coded in inline C.

Parameters:	p	The DCL_PI structure
	rk	The controller set-point reference
	yk	The measured feedback
Return:	The control effort	

DCL_runPI_C3 — Run the Parallel Form PI Controller

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runPI_C3(DCL_PI *p, float32_t rk, float32_t yk)

Description: This function executes a parallel form PI controller on the FPU32. The function is coded in inline C.

Parameters:	p	The DCL_PI structure
	rk	The controller set-point reference
	yk	The measured feedback
Return:	The control effort	

DCL_runPI_C4 — Run the Parallel Form PI Controller

Header File: DCLF32.h

Source File: DCL_PI_C4.asm

Declaration: float32_t DCL_runPI_C4(DCL_PI *p, float32_t rk, float32_t yk)

Description: This function executes a parallel form PI controller on the FPU32, and is identical in structure and operation to the C3 form. The function is coded in inline C.

Parameters:	p	The DCL_PI structure

	rk	The controller set-point reference
	yk	The measured feedback
Return:	The control effort	

DCL_runPI_C5 — Run the Parallel Form PI Controller

Header File: DCLF32.h

Source File: DCL_PI_C5.asm

Declaration: float32_t DCL_runPI_C5(DCL_PI *p, float32_t rk, float32_t yk)

Description: This function executes a parallel form PI controller on the FPU32. The configuration includes enhanced anti-windup reset logic which produces faster recovery from integral path saturation. Note that this controller cycle count is a little higher than C4. The function is coded in inline C.

Parameters:	p	The DCL_PI structure
	rk	The controller set-point reference
	yk	The measured feedback
Return:	The control effort	

DCL_runPI_L1 — Run the Ideal Form PI Controller

Header File: DCLCLA.h

Source File: DCL_PI_L1.asm

Declaration: float32_t DCL_runPI_L1(DCL_PI_CLA *p, float32_t rk, float32_t yk)

Description: This function executes an ideal form PI controller on the CLA. The function is coded in CLA assembly.

Parameters:	p	The DCL_PI_CLA structure
	rk	The controller set-point reference
	yk	The measured feedback
Return:	The control effort	

DCL_runPI_L2 — Run the Parallel Form PI Controller

Header File: DCLCLA.h

Source File: DCL_PI_L2.asm

Declaration: float32_t DCL_runPI_L2(DCL_PI_CLA *p, float32_t rk, float32_t yk)

Description: This function executes a parallel form PI controller on the CLA. The function is coded in CLA assembly.

Parameters:

p	The DCL_PI_CLA structure
rk	The controller set-point reference
yk	The measured feedback

Return: The control effort

DCL_runPI_L3 — ***Run the Ideal Form PI Controller***

Header File: DCLCLA.h

Source File: N/A

Declaration: float32_t DCL_runPI_L3(DCL_PI_CLA *p, float32_t rk, float32_t yk)

Description: This function executes an ideal form PI controller on the CLA. The function is coded in C.

Parameters:

p	The DCL_PI_CLA structure
rk	The controller set-point reference
yk	The measured feedback

Return: The control effort

DCL_runPI_L4 — ***Run the Parallel Form PI Controller***

Header File: DCLCLA.h

Source File: N/A

Declaration: float32_t DCL_runPI_L4(DCL_PI_CLA *p, float32_t rk, float32_t yk)

Description: This function executes a parallel form PI controller on the CLA. The function is coded in C.

Parameters:

p	The DCL_PI_CLA structure
rk	The controller set-point reference
yk	The measured feedback

Return: The control effort

DCL_resetPI	***Resets the PI Controller***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_resetPI(DCL_PI *p)

Description: This function resets the internal variables in the DCL_PI structure to default values. The integrator accumulator is set to zero, and the `err` field in the CSS sub-structure is set NONE. Note that the function is atomic.

Parameters: p The DCL_PI structure

Return: Void

DCL_updatePI	***Updates the PI Controller Parameters***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_updatePI(DCL_PI *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PI structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_PI structure

Return: Void

DCL_fupdatePI	***Updates the PI Controller Parameters***

Header File: DCLF32.h

Source File: DCL_futils.asm

Declaration: void DCL_fupdatePI(DCL_PI *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PI structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters. The function is implemented as an assembly module.

Parameters: p The DCL_PI structure

Return: Void

DCL_loadSeriesPIasZPK	**Load the Series Form PI Controller from ZPK**

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadSeriesPIasZPK(DCL_PI *p, DCL_ZPK3 *q)

Description: Loads the SPS coefficients to implement a series form PI controller (e.g. C1, C2 or L1) based on a ZPK3 definition (one real zero plus integrator). Note that the active coefficients are not affected until the controller is updates using `DCL_updatePI()`. Refer to section 1.3.6 for more information on the ZPK3 structure.

Parameters: p The DCL_PI structure

q The ZPK3 structure

Return: Void

DCL_loadParallelPIasZPK	**Load the Parallel Form PI Controller from ZPK**

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadSeriesPIasZPK(DCL_PI *p, DCL_ZPK3 *q)

Description: Loads the SPS coefficients to implement a parallel form PI controller (e.g. C3, C4, C5, or L2) based on a ZPK3 definition (one real zero plus integrator). Note that the active coefficients are not affected until the controller is updates using `DCL_updatePI()`. Refer to section 1.3.6 for more information on the ZPK3 structure.

Parameters: p The DCL_PI structure

q The ZPK3 structure

Return: Void

3.3 Non-linear PID Controller

3.3.1 Description

The DCL includes two implementations of a non-linear PID controller, denoted NLPID. The controllers are broadly similar to the ideal PID_C1 controller, except that there is no set-point weighting and in one case the derivative path sees the servo error instead of the feedback. A non-linear gain block appears in series with each of the three paths.

To improve computational efficiency, the non-linear law is separated into two parts: one part which is common to all paths, and a second part which contains terms specific to each path. The non-linear part of the high-level controller structure is shown below:

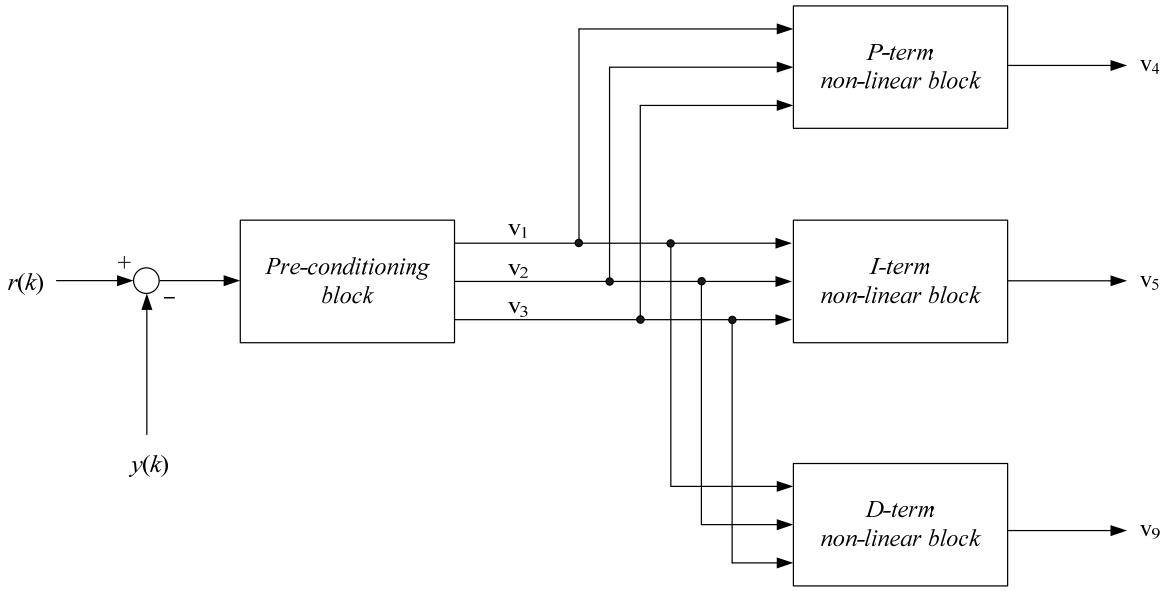


Figure 12. DCL_NLPID C1 input architecture

The linear part of the DCL_NLPID_C1 controller is shown below:

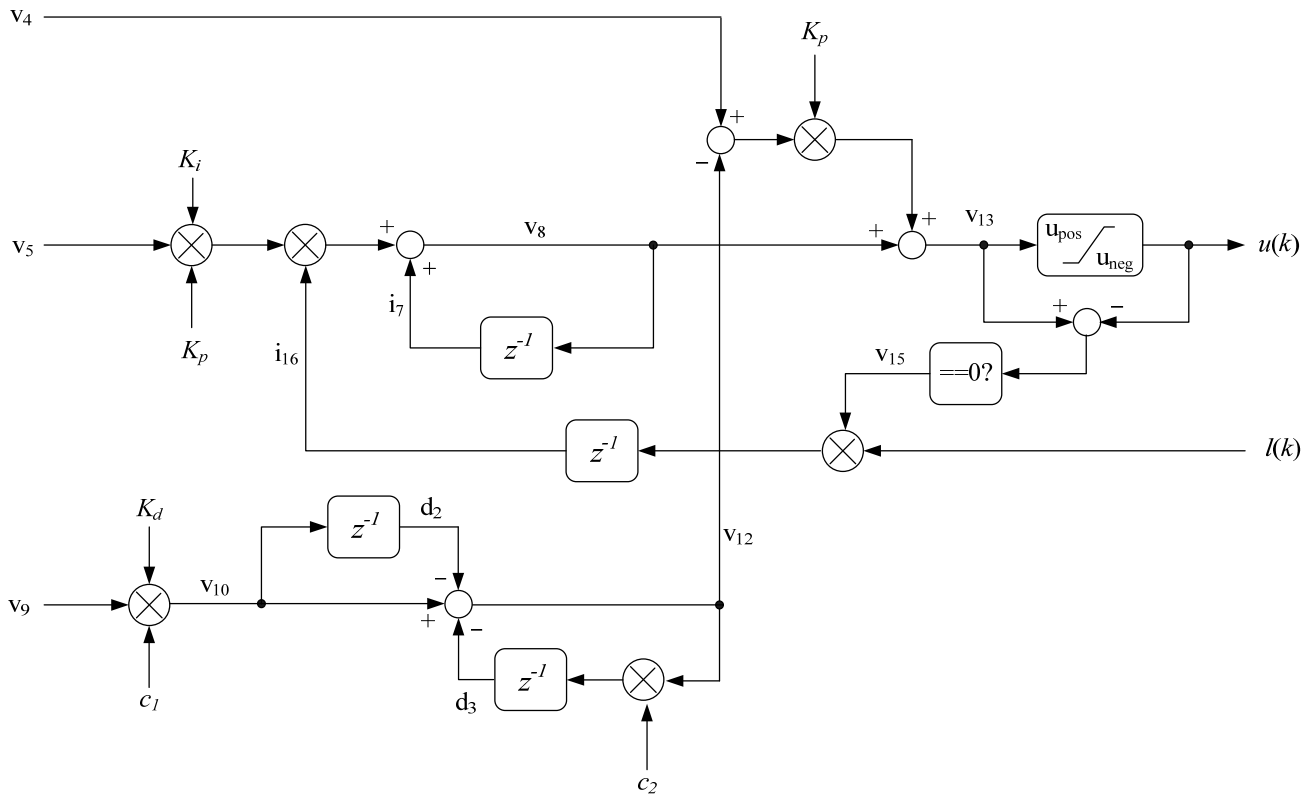


Figure 13. DCL_NLPID C1 output architecture

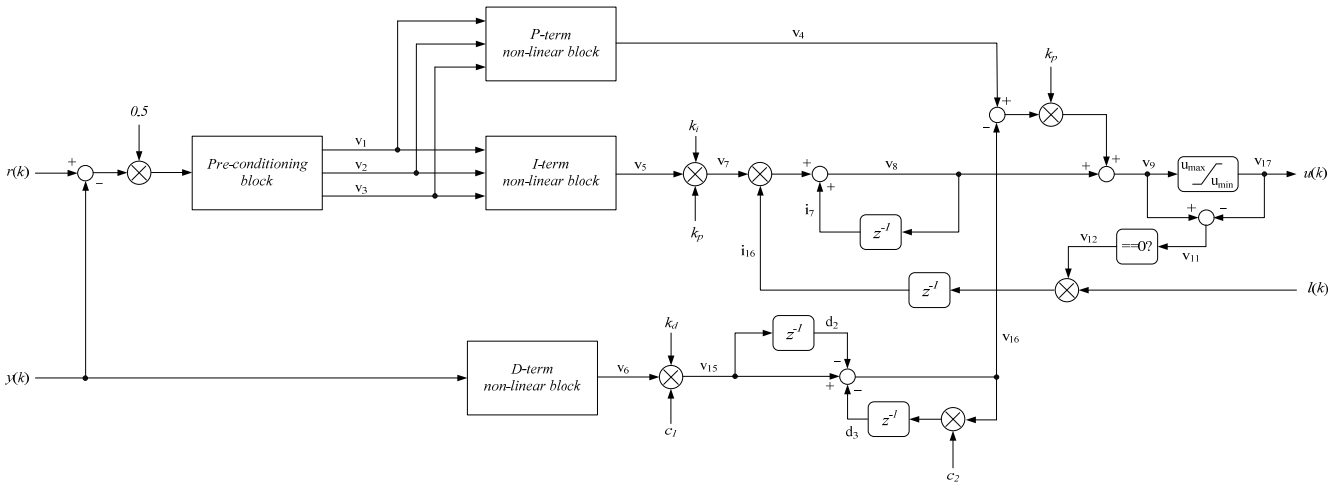


Figure 14. DCL_NLPID C2 output architecture

The non-linear control law is based on a power function of the modulus of the servo error, with a linearized region about the zero error point. In the equation below, x represents the input to the control law, y, the output, and α is a user selectable modulus exponent representing the degree of non-linearity.

Equation 9. $y = |x|^{\alpha} sign(x)$

The following plot shows the x-y relationship for values of α between 0.2 and 2. Notice that the curves intersect at x = 0, and x = ±1.

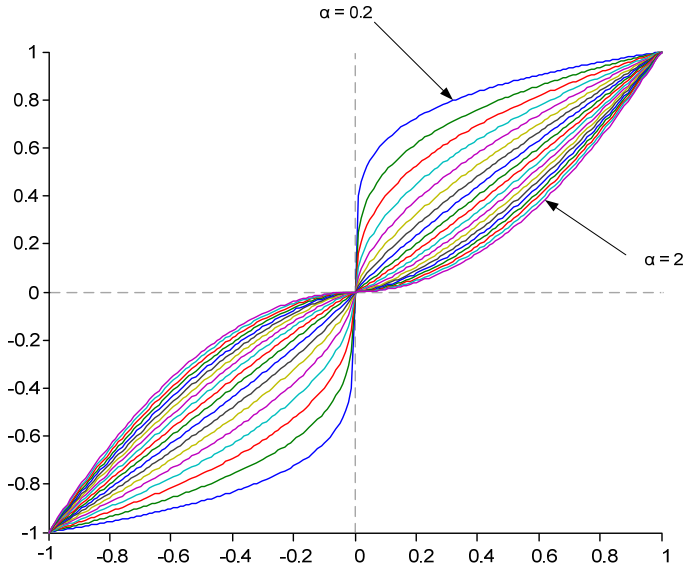


Figure 15. Non-linear control law input-output plot

The gain of the control law is the slope of the x-y curve. Observe that with α = 1 the control is linear with unity gain. With α > 1 the gain is zero when x = 0, and increases as x increases. In the controller, a value of α in this range produces controller gain which increases with increasing control error. With 0 < α < 1 the gain at x = 0 is infinite, and falls as x increases. This range of α setting produces controller gain which decreases with increasing servo error.

The plots below show the gain vs. control loop error curves for different values of α. Notice particularly the singularity at x = 0 when α < 1.

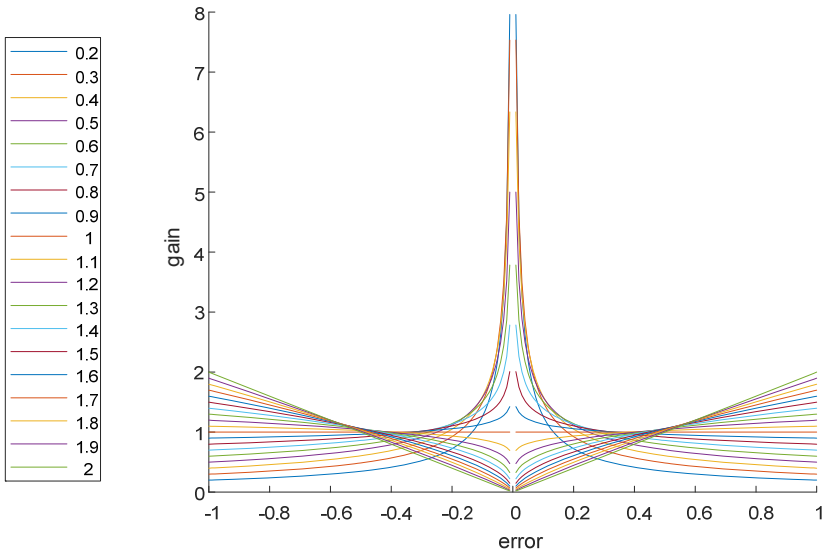


Figure 16. *Gain vs error curves for varying alpha*

The presence of zero or infinite gain at the zero control error point leads to practical difficulties. With α > 1 the response becomes sluggish for small errors; with α < 1 it is common to encounter oscillation or "chattering" near steady-state. These issues can be overcome by limiting the controller gain in a region close to x = 0. This is done by modifying the control law to introduce a user selectable region about x = 0 with linear gain is applied. The non-linear control law becomes

Equation 10.
$$y = \begin{cases} |x|^{\alpha} sign(x) & : |x| \geq \delta \\ x\,\delta^{\alpha-1} & : |x| \leq \delta \end{cases}$$

When the magnitude of servo error falls below δ the linear gain is applied, otherwise the gain is determined by the non-linear law. For computational efficiency, we will define the gain in the linear region as γ.

Equation 11. $\gamma = \delta^{\alpha-1}$

A typical plot of the linearized control law is shown below. Observe that when x = δ the linear and non-linear curves intersect, so the controller makes a smooth transition between the linear and non-linear regions as the servo error passes through x = ±δ.

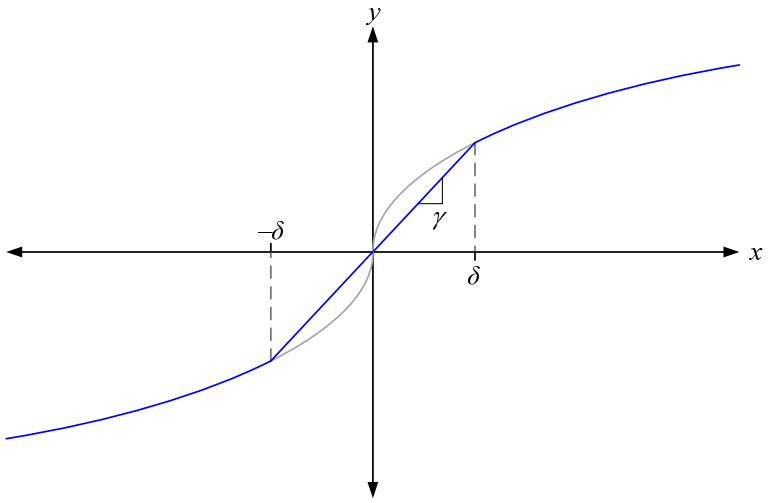


Figure 17. NLPID linearized region

In addition to the P, I, and D gains, the user must select two additional terms in each control path: α and δ. The library includes a separate function to compute and update γ for each path using the SPS structure. The use must initialize γ parameters before calling the NL controllers.

The NLPID controller has been seen to provide significantly improved control in many cases, however it must be remembered that increased gain, even if only applied to part of the control range, can lead to significantly increased output from the controller. In most cases, this 'control effort' is limited by practical factors such as actuator saturation, PWM modulation range, and so on. The corollary is that not every application benefits equally from the use of non-linear control and some may see no benefit at all. In cases where satisfactory performance cannot be achieved through the use of linear PID control, the user is advised to start with all α = 1, and experiment by introducing non-linear terms gradually while monitoring both control performance and the magnitude of the control effort. In general, P and I paths benefit from increased gain at high servo error (α > 1), while the D path benefits from reduced gain at high servo error (α < 1), but this is not universally true.

Further information on this control law can be found in: "*From PID to Active Disturbance Rejection Control*", Jingqing Han, IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, VOL. 56, NO. 3, MARCH 2009

3.3.2 Implementation

The NLPID controller uses a C structure to hold coefficients and data, defined in the header file `DCL_NLPID.h`. Note that the NLPID functions make use of the `pow()` function in the standard C library. For this reason the header file `math.h` must be included, which is not supported by the CLA compiler. To allow different DCL functions to be run on both the CPU and CLA in the same program, the NLPID functions are located in a separate header file. Refer to the file `DCL_NLPID.h` for details of the NLPID controller structure.

As with all DCL controllers, it is the responsibility of the user to initialize the DCL_NLPID structure before use. A set of default values is defined in the library header file and can

be used with the variable declaration. An example of an initialized DCL_NLPID structure declaration is shown below.

```
DCL_NLPID myCtrl = NLPID_DEFAULTS;
```

3.3.3 Functions

DCL_runNLPID_C1	***Run the Non-linear PID Controller***

Header File: DCL_NLPID.h

Source File: N/A

Declaration: float32_t DCL_runNLPID_C1(DCL_NLPID *p, float32_t rk, float32_t yk,
float32_t lk)

Description: This function executes a parallel form non-linear PID controller on the FPU32. The function is coded in inline C.

Parameters:

p	The DCL_NLPID structure
rk	The controller set-point reference
yk	The measured feedback
lk	External output clamp flag

Return: The control effort

DCL_runNLPID_C2	***Run the Non-linear PID Controller***

Header File: DCL_NLPID.h

Source File: N/A

Declaration: float32_t DCL_runNLPID_C2(DCL_NLPID *p, float32_t rk, float32_t yk,
float32_t lk)

Description: This function executes a series form non-linear PID controller on the FPU32. This controller is broadly similar to C1 except that the derivative path feedback comes from the loop output rather than the error. The function is coded in inline C.

Parameters:

p	The DCL_NLPID structure
rk	The controller set-point reference
yk	The measured feedback
lk	External output clamp flag

Return: The control effort

DCL_setGamma	***Compute the Non-linear PID Gain Limits***

Header File: DCL_NLPID.h

Source File: N/A

Declaration: void DCL_setGamma(DCL_NLPID *p)

Description: This function computes the three gain limits for the non-linear PID controller on the C28x. The function is coded in inline C.

Parameters: p The DCL_NLPID structure

Return: The control effort

DCL_resetNLPID	***Reset the NLPID Controller***

Header File: DCL_NLPID.h

Source File: N/A

Declaration: void DCL_resetNLPID(DCL_NLPID *p)

Description: This function resets the internal variables in the DCL_NLPID structure to default values. The integrator accumulator and store derivative path values are set to 0.0, and the integrator clamp variable set to 1.0. The function also sets the `err` field in the CSS sub-structure to NONE. Note that the function is atomic.

Parameters: p The DCL_NLPID structure

Return: Void

DCL_updateNLPID	***Update the NLPID Controller Parameters***

Header File: DCL_NLPID.h

Source File: N/A

Declaration: void DCL_updateNLPID(DCL_NLPID *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PID structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_NLPID structure

Return: Void

DCL_setActiveNLPIDfilterBW — Set the Active NLPID Derivative Filter Bandwidth

Header File: DCL_NLPID.h

Source File: N/A

Declaration: void DCL_setActiveNLPIDfilterBW(DCL_NLPID *p, float32_t fc, float32_t T)

Description: Loads the active derivative filter coefficients `c1` & `c2` in the NLPID structure based on the desired filter bandwidth specified in Hz and controller update rate in seconds. Coefficients in the SPS and CSS are unaffected.

Parameters:

p	The DCL_NLPID structure
fc	The desired filter bandwidth in Hz
T	The controller update rate in seconds

Return: Void

DCL_setNLPIDfilterBW — Set the NLPID Derivative Filter Bandwidth

Header File: DCL_NLPID.h

Source File: N/A

Declaration: void DCL_setNLPIDfilterBW(DCL_NLPID *p, float32_t fc)

Description: Loads the derivative filter coefficients `c1` & `c2` in the SPS based on the desired filter bandwidth specified in Hz. Coefficients in the active parameter set are unaffected until the controller is updated using DCL_updateNLPID().

Parameters:

p	The DCL_NLPID structure
fc	The desired filter bandwidth in Hz

Return: Void

DCL_getNLPIDfilterBW — Get the NLPID Derivative Filter Bandwidth

Header File: DCL_NLPID.h

Source File: N/A

Declaration: float32_t DCL_getNLPIDfilterBW(DCL_NLPID *p)

Description: Finds the bandwidth of the current derivative filter in Hz by examining the coefficients in the active parameter set (i.e. not the SPS).

Parameters:

p	The DCL_NLPID structure

Return: The active derivative filter bandwidth in Hz

DCL_getNLPIDgamma	***Get the NLPID Steady State Gain***

Header File: DCL_NLPID.h

Source File: N/A

Declaration: float32_t DCL_getNLPIDgamma(float32_t alpha, float32_t delta)

Description: Finds the steady state gain in the linearized region for the specified alpha & delta choice.

Parameters: p The DCL_NLPID structure

Return: The steady state gain

DCL_getNLPIDdelta	***Get the NLPID Linearized Region Semi-width***

Header File: DCL_NLPID.h

Source File: N/A

Declaration: float32_t DCL_getNLPIDdelta(float32_t alpha, float32_t gamma)

Description: Finds the semi-width of the linearized region from the specified alpha and gamma choice.

Parameters: p The DCL_NLPID structure

Return: The linearized region semi-width

3.4 Non-linear PI Controllers

3.4.1 Description

The DCL contains one non-linear PI controller, similar in form to DCL_PI_C1. Refer to section 3.3.1 for information on the non-linear control law.

3.4.2 Implementation

The NLPI controller is similar to a linear series form implementation, but with non-linear law blocks in the P and I paths. The controller uses a common C structure to hold coefficients and data, defined in the header files `DCL_NLPID.h` and `DCL.h`. The NLPI_C1 controller architecture is shown below.

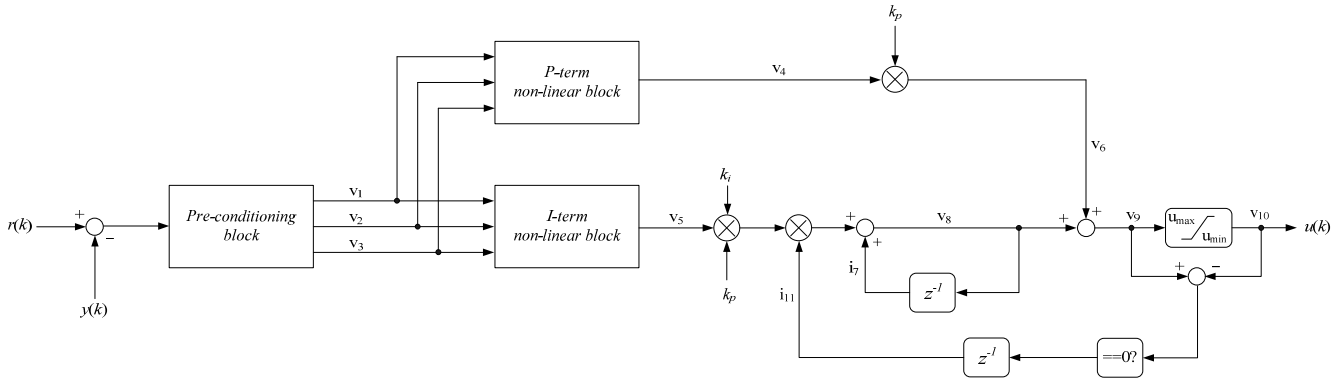


Figure 18. The DCL_NLPI_C1 architecture

3.4.3 Functions

DCL_runNLPI_C1	***Run the Ideal Form Non-linear PI Controller***

Header File: DCL_NLPID.h

Source File: N/A

Declaration: float32_t DCL_runNLPI_C1(DCL_NLPI *p, float32_t rk, float32_t yk)

Description: This function executes an ideal form PI controller on the FPU32. The function is coded in inline C.

Parameters:

p	The DCL_NLPI structure
rk	The controller set-point reference
yk	The measured feedback

Return: The control effort

DCL_resetNLPI	***Reset the Non-linear PI Controller***

Header File: DCL_NLPID.h

Source File: N/A

Declaration: void DCL_resetNLPI(DCL_NLPI *p)

Description: This function resets the internal dynamic variables in the DCL_NLPI structure to their default values. The integrator accumulator is set to zero, and the `err` field in the CSS sub-structure is set NONE. Note that this function is atomic.

Parameters:

p	The DCL_NLPI structure

Return: Void

DCL_updateNLPI	**Update the Non-linear PI Controller Parameters**

Header File: DCL_NLPID.h

Source File: N/A

Declaration: void DCL_updateNLPI(DCL_NLPI *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the control parameters stored in the SPS sub-structure are copied into the PI structure and `sts` is cleared. Dynamic variables are not affected. Note that this function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_NLPI structure

Return: Void

3.5 Double Integrator PI Controller

3.5.1 Description

The DCL contains one implementation of a linear PI controller having two series integrators. This type of controller is similar to the parallel PI controller described above except that the anti-windup reset logic is more complicated.

In this controller, allowance has been made for the `Kp` element to be zero. This scenario presents a problem if the controller enters saturation because without the proportional path there is no way to recover. The PI2 resolves this by releasing the anti-windup lock when the integrator input reverses sign. The logic has to be implemented twice, since there are two cascaded integrators. Similar anti-windup reset logic is present in the PI_C5 controller (see section 3.2 for more information).

3.5.2 Implementation

The double integrator PI2 controller uses a C structure to hold coefficients and data, defined in the header file `DCLF32.h`.

The PI2 implementation is shown below:

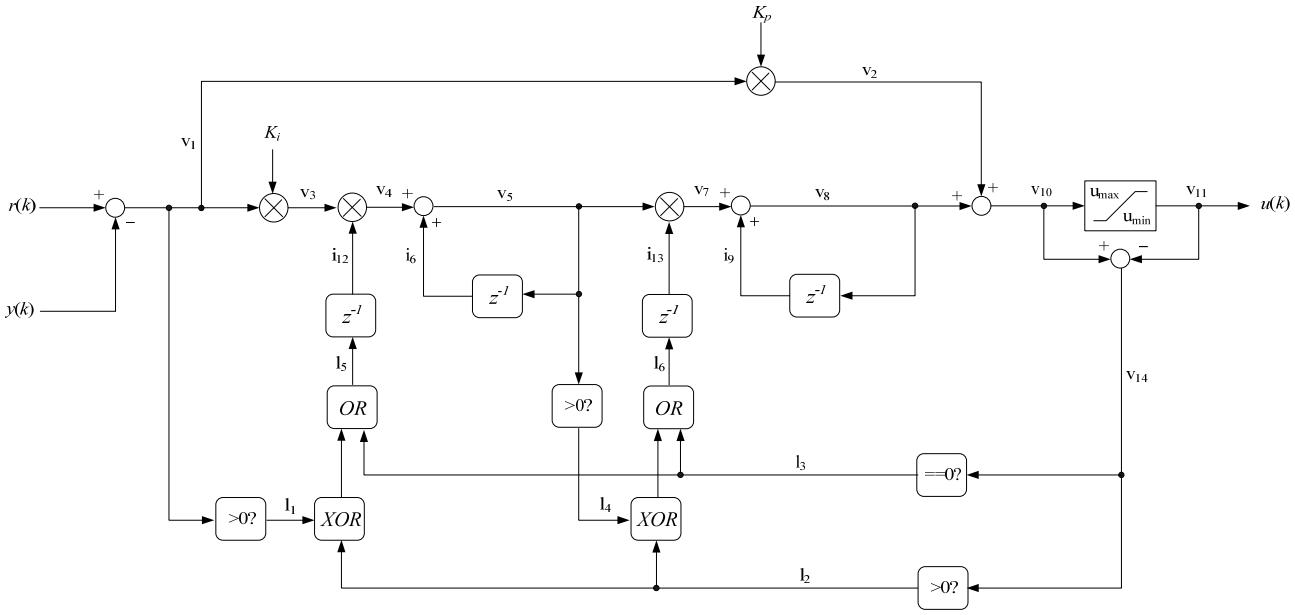


Figure 19. DCL_PI2 C1 architecture

3.5.3 Functions

DCL_runPI2_C1	**Run the Ideal Form PI2 Controller**

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runPI2_C1(DCL_PI2 *p, float32_t rk, float32_t yk)

Description: This function executes an ideal form PI2 controller on the FPU32. The function is coded in C.

Parameters:

	p	The DCL_PI2 structure
	rk	The controller set-point reference
	yk	The measured feedback

Return: The control effort

DCL_resetPI2	**Resets the PI2 Controller**

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_resetPI2(DCL_PI2 *p)

Description: This function resets the internal variables in the DCL_PI2 structure to default values. Both integrator accumulators are set to zero, and the `err` field in the CSS sub-structure is set NONE. Note that the function is atomic.

Parameters: p The DCL_PI2 structure

Return: Void

DCL_updatePI2	***Updates the PI2 Controller Parameters***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_updatePI2(DCL_PI2 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PI2 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_PI2 structure

Return: Void

DCL_fupdatePI2	***Updates the PI2 Controller Parameters***

Header File: DCLF32.h

Source File: DCL_futils.asm

Declaration: void DCL_fupdatePI2(DCL_PI2 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PI2 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters. The function is implemented as an assembly module.

Parameters: p The DCL_PI2 structure

Return: Void

3.6 Direct Form 1 (First Order) Compensators

3.6.1 Description

The DCL includes one first order compensator in Direct Form 1. The DF11 compensator implements a first order, or “simple lag”, type frequency response. The general form of discrete time first order transfer function is

Equation 12. $$F(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}$$

Denominator coefficients must be normalized accordingly. The corresponding difference equation is

Equation 13. $$u(k) = b_0 e(k) + b_1 e(k-1) - a_1 u(k-1)$$

A diagrammatic representation of the DF11 is shown below:

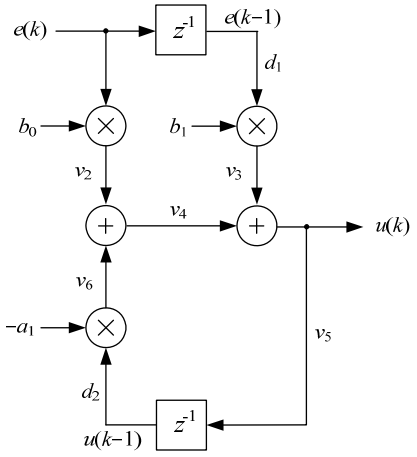


Figure 20. DCL_DF11 C1, C2, & L1 architecture

3.6.2 Implementation

All DF11 functions use a common C structure to hold coefficients and data, defined in the header files `DCLF32.h` and `DCLCLA.h`.

It is the responsibility of the user to initialize coefficients and data prior to use. A set of default values is defined in the library header file and can be used with the variable declaration. An example of an initialized DCL_DF11 structure declaration on FPU32 is shown below:

```
DCL_DF11 myCtrl = DF11_DEFAULTS;
```

3.6.3 Functions

DCL_runDF11_C1	***Run the DF11 Compensator***

Header File: DCLF32.h

Source File: DCL_DF11_C1.asm

Declaration: float32_t DCL_runDF11_C1(DCL_DF11 *p, float32_t ek)

Description: This function computes a first order control law using the Direct Form 1 structure. The function is coded in FPU32 assembly.

Parameters: p The DCL_DF11 structure

ek The servo error

Return: The control effort

DCL_runDF11_C2 — ***Run the DF11 Compensator***

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runDF11_C1(DCL_DF11 *p, float32_t ek)

Description: This function computes a first order control law using the Direct Form 1 structure. The function is coded in C.

Parameters: p The DCL_DF11 structure

ek The servo error

Return: The control effort

DCL_runDF11_L1 — ***Run the DF11 Compensator***

Header File: DCLCLA.h

Source File: DCL_DF11_L1.asm

Declaration: float32_t DCL_runDF11_L1(DCL_DF11_CLA *p, float32_t ek)

Description: This function computes a first order control law using the Direct Form 1 structure. The function is coded in CLA assembly.

Parameters: p The DCL_DF11_CLA structure

ek The servo error

Return: The control effort

DCL_resetDF11 — ***Resets the DF11 Compensator***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_resetDF11(DCL_DF11 *p)

Description: This function resets the internal variables in the DCL_DF11 structure to default values. The forward and return path coefficients are configured to implement a unity gain response, and the `err` field in the CSS sub-structure is set NONE. Note that the function is atomic.

Parameters: p The DCL_DF11 structure

Return: Void

DCL_updateDF11 — ***Updates the DF11 Compensator Parameters***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_updateDF11(DCL_DF11 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the DF11 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_DF11 structure

Return: Void

DCL_fupdateDF11 — ***Updates the DF11 Controller Parameters***

Header File: DCLF32.h

Source File: DCL_futils.asm

Declaration: void DCL_fupdateDF11(DCL_DF11 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the DF11 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters. The function is implemented as an assembly module.

Parameters: p The DCL_DF11 structure

Return: Void

DCL_isStableDF11 — ***Determines whether the DF11 Compensator is Stable***

Header File: DCLF32.h

Source File: N/A

Declaration: int16_t DCL_isStableDF11(DCL_DF11 *p)

Description: This function determines whether the coefficient set in the SPS sub-structure represent a stable compensator. If the pole magnitude is less than one, the function returns '1', indicating stability; otherwise the function returns '0'. Refer to section 1.3.7 for more information on compensator stability tests.

Parameters: p The DCL_DF11 structure

Return: '1' if stable, otherwise '0'

DCL_loadDF11asZPK — ***Loads the DF11 Compensator from ZPK***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadDF11asZPK (DCL_DF11 *p, DCL_ZPK3 *q)

Description: This function loads the DF11 compensator coefficients in the SPS sub-structure from a 1-pole, 1-zero description held in a ZPK3 structure. Active coefficients are unaffected until the `DCL_updateDF11()` function is called. Refer to section 1.3.6 for more information on the ZPK3 structure.

Parameters: p The DCL_DF11 structure

q The DCL_ZPK3 structure

Return: Void

DCL_loadDF11asPI — ***Loads the DF11 Compensator from a PI Description***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadDF11asPI (DCL_DF11 *p, float32_t Kp, float32_t Ki)

Description: This function loads the DF11 compensator coefficients in the SPS sub-structure to emulate a series form PI controller. Active coefficients are unaffected until the `DCL_updateDF11()` function is called.

Parameters: p The DCL_DF11 structure

Kp The equivalent series form PI controller proportional gain

Ki The equivalent series form PI controller integral gain

Return: Void

3.7 Direct Form 1 (Third Order) Compensators

3.7.1 Description

The Direct Form 1 (DF1) structure is a common type of discrete time control structure used to implement a control law or dynamical system model specified either as a pole-zero set, or as a rational polynomial in z (i.e. a discrete time transfer function). The DCL includes one third order DF1 compensator, denoted “DF13”.

In general, the Direct Form 1 structure is less numerically robust than the Direct Form 2 (see below), and for this reason users are encouraged to choose the latter type whenever possible. However, the DCL_DF13 structure is very common in digital power supplies and for that reason is included in the library. The same function supports a second order control law after the superfluous coefficients (a_3 & b_3) have been set to zero.

The general form of third order transfer function is

Equation 14.
$$F(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + b_3 z^{-3}}{1 + a_1 z^{-1} + a_2 z^{-2} + a_3 z^{-3}}$$

Notice that the coefficients have been adjusted to normalize the highest power of z in the denominator. There is no notational standard for numbering of the controller coefficients; the notation used here has the advantage that the coefficient suffixes are the same as the delay line elements and this helps with clarity of the assembly code, however other notations may be found in the literature. The corresponding difference equation is

Equation 15.
$$u(k) = b_0 e(k) + b_1 e(k-1) + b_2 e(k-2) + b_3 e(k-3) - a_1 u(k-1) - a_2 u(k-2) - a_3 u(k-3)$$

The DF13 controller uses two, three-element delay lines to store previous input and output data required to compute u(k). A diagrammatic representation is shown below.

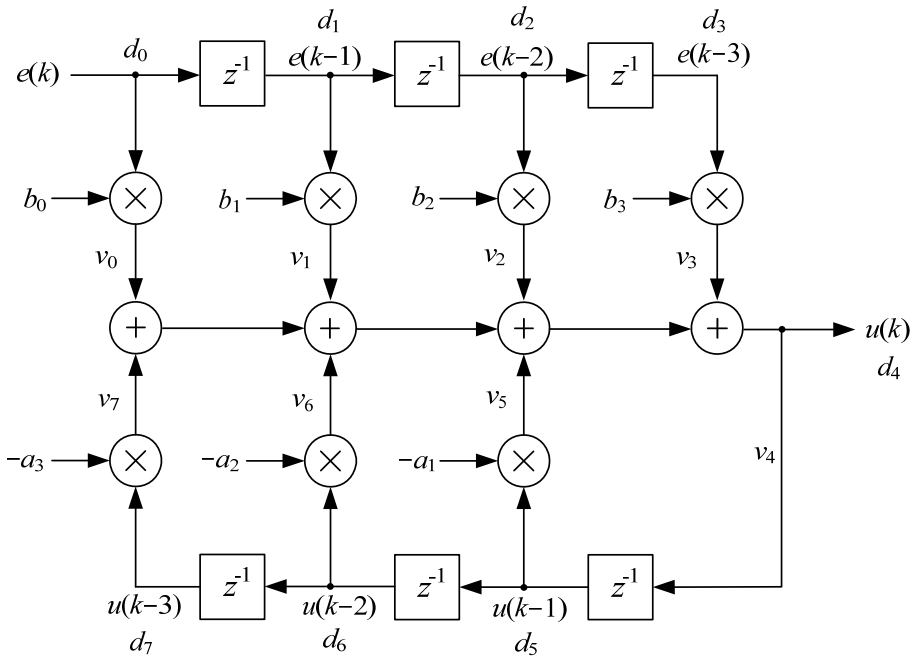


Figure 21. *DCL_DF13 C1, C4, & L1 architecture*

The DF13 control law consists of seven multiplication operations which yield seven partial products, and six addition or subtraction operations which combine the partial products to obtain the compensator output, u(k). When implemented in this way, the control law is referred to as the “full” DF13 form.

The DF13 control law can be re-structured to reduce control latency by pre-computing six of the seven partial products which are already known in the previous sample interval. The control law is then broken into two parts: the “immediate” part and the “partial” part.

The advantage of doing this is to reduce the “sample-to output” delay, or the time between e(k) being sampled, and a corresponding u(k) becoming available. By partially pre-computing the control law, the computation delay can be reduced to one multiplication and one addition.

In the k^{th} interval, the immediate part is computed.

Equation 16. $u(k) = b_0 e(k) + v(k-1)$

Next, the v(k) partial result is pre-computed for use in the $(k+1)^{th}$ interval.

Equation 17. $v(k) = b_1 e(k) + b_2 e(k-1) + b_3 e(k-2) - a_1 u(k) - a_2 u(k-1) - a_3 u(k-2)$

Structurally, the pre-computed control law can be drawn as below:

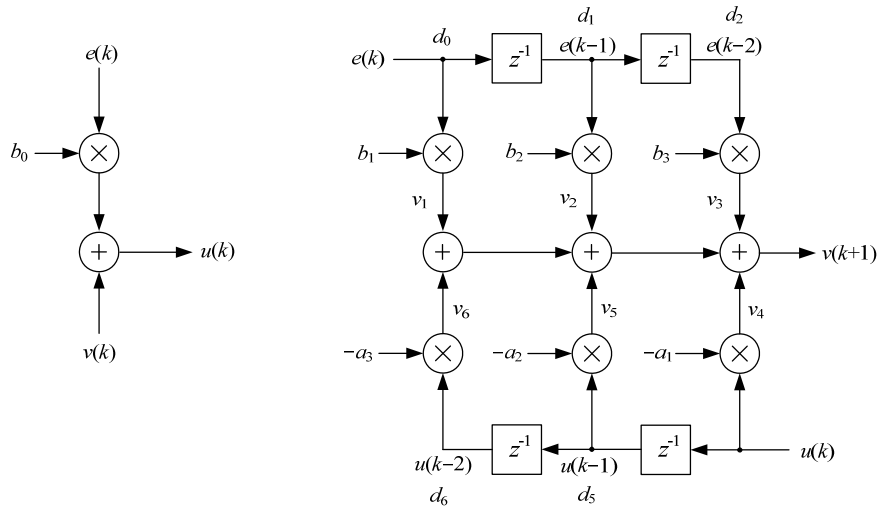


Figure 22. *DCL_DF13 C2, C3, C5, C6, L2, & L3 architecture*

The pre-computed structure allows the controller output (u(k)) to be used as soon as it is computed. The remaining terms in the third order control law do not involve the newest input e(k) and therefore do not affect u(k). These terms can be computed after u(k) has been applied to the control loop and the input-output latency of the controller is therefore reduced.

A further benefit of the pre-computed structure is that it allows the control effort to be clamped after the immediate part. Computation of the pre-computed part can be made dependent on the outcome of the clamp such that if u(k) matches or exceeds the clamp limits there is no point in pre-computing the next partial control variable and the computation can be avoided. The DCL includes three clamp functions intended for this purpose (see chapter 6).

3.7.2 Implementation

All DF13 functions use a common C structure to hold coefficients and data, defined in the header files `DCL.h` and `DCLCLA.h`.

The assignment of coefficients and data in the DCL_DF13 structure to those in the diagram is shown below:

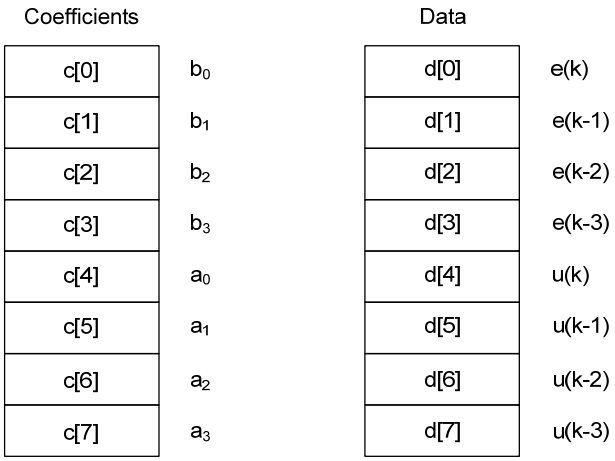


Figure 23. DCL_DF13 data & coefficient layout

It is the responsibility of the user to initialize both arrays prior to use. A set of default values is defined in the library header file and can be used with the variable declaration. An example of an initialized DF13 structure declaration is shown below:

```
DCL_DF13 myCtrl = DF13_DEFAULTS;
```

3.7.3 Functions

DCL_runDF13_C1	***Run the DF13 Full Compensator***

Header File: DCLF32.h

Source File: DCL_DF13_C1.asm

Declaration: float32_t DCL_runDF13_C1(DCL_DF13 *p, float32_t ek)

Description: This function computes a full third order control law using the Direct Form 1 structure. The function is coded in FPU32 assembly.

Parameters:

p	The DCL_DF13 structure
ek	The servo error

Return: The control effort

DCL_runDF13_C2	***Run the Immediate DF13 Compensator***

Header File: DCLF32.h

Source File: DCL_DF13_C2C3.asm

Declaration: float32_t DCL_runDF13_C2(DCL_DF13 *p, float32_t ek, float32_t vk)

Description: This function computes the immediate part of the pre-computed DF13 controller. The function is coded in FPU32 assembly.

Parameters:
- p — The DCL_DF13 structure
- ek — The servo error
- vk — The pre-computed partial control effort

Return: The control effort

DCL_runDF13_C3 — ***Run the Partial DF13 Compensator***

Header File: DCLF32.h

Source File: DCL_DF13_C2C3.asm

Declaration: float32_t DCL_runDF13_C3(DCL_DF13 *p, float32_t ek, float32_t uk)

Description: This function computes the partial result of the pre-computed DF13 controller. The function is coded in FPU32 assembly.

Parameters:
- p — The DCL_DF13 structure
- ek — The servo error
- uk — The control effort in the previous sample interval

Return: The control effort

DCL_runDF13_C4 — ***Run the DF13 Full Compensator***

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runDF13_C4(DCL_DF13 *p, float32_t ek)

Description: This function computes a full third order control law using the Direct Form 1 structure, and is identical in structure and operation to the C1 form. The function is coded in inline C.

Parameters:
- p — The DCL_DF13 structure
- ek — The servo error

Return: The control effort

DCL_runDF13_C5 — ***Run the Immediate DF13 Compensator***

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runDF13_C5(DCL_DF13 *p, float32_t ek, float32_t vk)

Description: This function computes the immediate part of the pre-computed DF13 controller. The function is identical in structure and operation to the C2 form. The function is coded in inline C.

Parameters:

p	The DCL_DF13 structure
ek	The servo error
vk	The pre-computed partial control effort

Return: The control effort

DCL_runDF13_C6 — ***Run the Partial DF13 Compensator***

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runDF13_C6(DCL_DF13 *p, float32_t ek, float32_t uk)

Description: This function computes the partial result of the pre-computed DF13 controller. The function is identical in structure and operation to the C3 form. The function is coded in inline C.

Parameters:

p	The DCL_DF13 structure
ek	The servo error
uk	The control effort in the previous sample interval

Return: The control effort

DCL_runDF13_L1 — ***Run the DF13 Full Compensator on the CLA***

Header File: DCLCLA.h

Source File: DCL_DF13_L1.asm

Declaration: float32_t DCL_runDF13_L1(DCL_DF13_CLA *p, float32_t ek)

Description: This function computes a full third order control law using the Direct Form 1 structure, and is identical in structure and operation to the C1 form. The function is coded in CLA assembly language.

Parameters:

p	The DCL_DF13_CLA structure
ek	The servo error

Return: The control effort

DCL_runDF13_L2 — ***Run the Immediate DF13 Compensator on the CLA***

Header File: DCLCLA.h

Source File: DCL_DF13_L2L3.asm

Declaration: float32_t DCL_runDF13_L2(DCL_DF13_CLA *p, float32_t ek, float32_t vk)

Description: This function computes the immediate part of the pre-computed DF13 controller. The function is identical in structure and operation to the C2 form. The function is coded in CLA assembly language.

Parameters:

p	The DCL_DF13_CLA structure
ek	The servo error
vk	The pre-computed partial control effort

Return: The control effort

DCL_runDF13_L3 — ***Run the Partial DF13 Compensator on the CLA***

Header File: DCLCLA.h

Source File: DCL_DF13_L2L3.asm

Declaration: float32_t DCL_runDF13_L3(DCL_DF13_CLA *p, float32_t ek, float32_t uk)

Description: This function computes the partial result of the pre-computed DF13 controller. The function is identical in structure and operation to the C3 form. The function is coded in CLA assembly language.

Parameters:

p	The DCL_DF13_CLA structure
ek	The servo error
uk	The control effort in the previous sample interval

Return: The control effort

DCL_resetDF13 — ***Resets the DF13 Compensator***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_resetDF13(DCL_DF13 *p)

Description: This function resets the internal variables in the DCL_DF13 structure to default values. The forward and return path coefficients are configured to implement a unity gain response, and the `err` field in the CSS sub-structure is set NONE. Note that the function is atomic.

Parameters:

p	The DCL_DF13 structure

Return: Void

DCL_updateDF13 — ***Updates the DF13 Compensator Parameters***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_updateDF13(DCL_DF13 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the DF13 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_DF13 structure

Return: Void

DCL_fupdateDF13 — ***Updates the DF13 Controller Parameters***

Header File: DCLF32.h

Source File: DCL_futils.asm

Declaration: void DCL_fupdateDF13(DCL_DF13 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the DF13 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters. The function is implemented as an assembly module.

Parameters: p The DCL_DF13 structure

Return: Void

DCL_isStableDF13 — ***Determines whether the DF13 Compensator is Stable***

Header File: DCLF32.h

Source File: N/A

Declaration: int16_t DCL_isStableDF13(DCL_DF13 *p)

Description: This function determines whether the coefficient set in the SPS sub-structure represent a stable compensator. If the pole magnitude is less than one, the function returns ‘1’, indicating stability; otherwise the function returns ‘0’. Refer to section 1.3.7 for more information on compensator stability tests.

Parameters: p The DCL_DF13 structure

Return: ‘1’ if stable, otherwise ‘0’

DCL_loadDF13asZPK — ***Loads the DF13 Compensator from ZPK***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadDF13asZPK (DCL_DF13 *p, DCL_ZPK3 *q)

Description: This function loads the DF13 compensator coefficients in the SPS sub-structure from a 3-pole, 3-zero description held in a ZPK3 structure. Active coefficients are unaffected until the `DCL_updateDF13()` function is called. Refer to section 1.3.6 for more information on the ZPK3 structure.

Parameters:

	p	The DCL_DF13 structure
	q	The DCL_ZPK3 structure

Return: Void

3.8 Direct Form 2 (Second Order) Compensators

3.8.1 Description

The C2000 Digital Controller Library contains a second order implementation of the Direct Form 2 controller structure, denoted “DCL_DF22”. This structure is sometimes referred to as a “bi-quad” filter and is commonly used in a cascaded chain to build up digital filters of high order.

The transfer function of a second order discrete time compensator is

Equation 18. $$F(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

The corresponding difference equation is

Equation 19. $$u(k) = b_0 e(k) + b_1 e(k-1) + b_2 e(k-2) - a_1 u(k-1) - a_2 u(k-2)$$

A diagrammatic representation of the full Direct Form 2 realization is shown below:

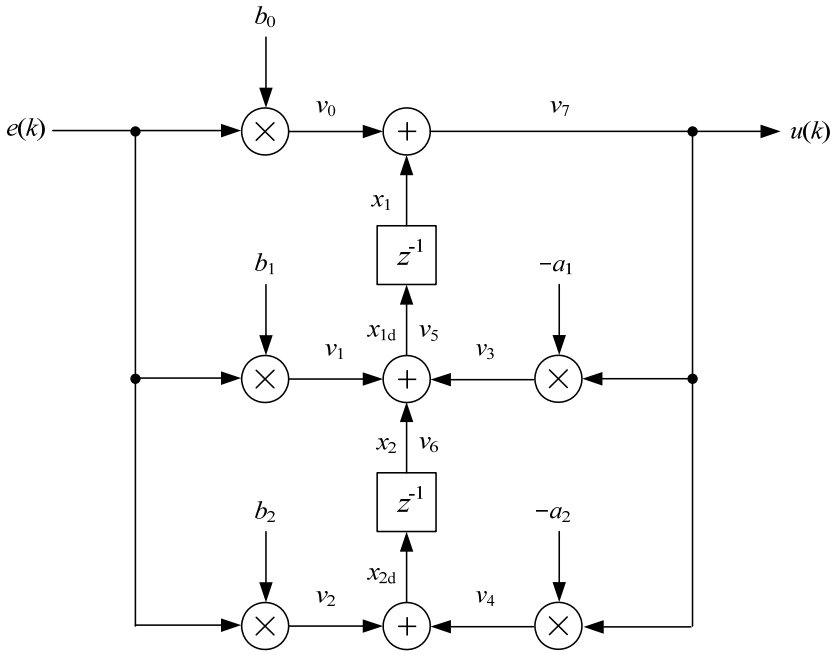


Figure 24. DCL_DF22 C1, C4,L1, & L4 architecture

As with the DCL_DF13 compensator, sample-to-output delay can be reduced through the use of pre-computation. The immediate and pre-computed control laws are as follows. In the k[th] interval, the immediate part is computed.

Equation 20. $u(k) = b_0 e(k) + v(k)$

Next, the v(k) partial result is pre-computed for use in the (k+1)[th] interval.

Equation 21. $v(k+1) = b_1 e(k) + b_2 e(k-1) - a_1 u(k) - a_2 u(k-1)$

The pre-computed form of DCL_DF22 is shown in the following diagrams.

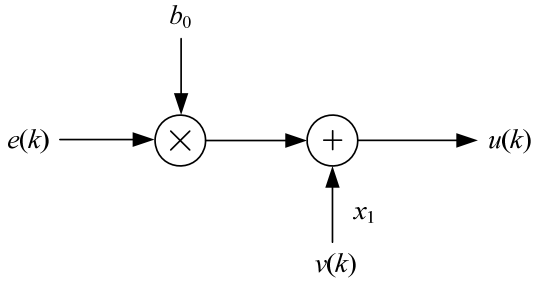


Figure 25. DCL_DF22 C2, C5, & L2 architecture

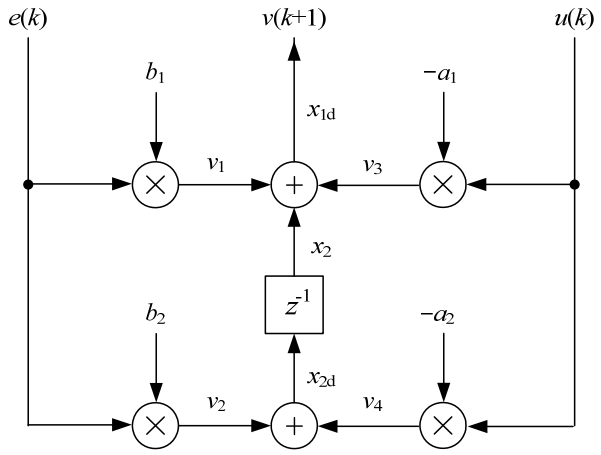


Figure 26. DCL_DF22 C3, C6, & L3 architecture

Notice that pre-computation is a little different from the Direct Form 1 case because the intermediate value exists as one of the internal states and is therefore automatically stored as “x1” in the DCL_DF22 structure. Therefore it is not necessary to create a separate variable to store v(k).

3.8.2 Implementation

All DF22 functions use a common C structure to hold coefficients and data, defined in the header file `DCL.h` and `DCLCLA.h`.

It is the responsibility of the user to initialize both arrays prior to use. A set of default values is defined in the library header file and can be used with the variable declaration. An example of an initialized DCL_DF22 structure declaration is shown below:

```
DF22 myCtrl = DF22_DEFAULTS;
```

3.8.3 Functions

DCL_runDF22_C1	***Run the DF22 Full Compensator***

Header File: DCLF32.h

Source File: DCL_DF22_C1.asm

Declaration: float32_t DCL_runDF22_C1(DCL_DF22 *p, float32_t ek)

Description: This function computes a full second order control law using the Direct Form 2 structure. The function is coded in FPU32 assembly.

Parameters:

p	The DCL_DF22 structure
ek	The servo error

Return: The control effort

DCL_runDF22_C2 — Run the Immediate DF22 Compensator

Header File: DCLF32.h

Source File: DCL_DF22_C2C3.asm

Declaration: float32_t DCL_runDF22_C2(DCL_DF22 *p, float32_t ek)

Description: This function computes the immediate part of the pre-computed DF22 controller. The function is coded in FPU32 assembly.

Parameters:

p	The DCL_DF22 structure
ek	The servo error

Return: The control effort

DCL_runDF22_C3 — Run the Partial DF22 Compensator

Header File: DCLF32.h

Source File: DCL_DF22_C2C3.asm

Declaration: void DCL_runDF22_C3(DCL_DF22 *p, float32_t ek, float32_t uk)

Description: This function computes the partial result of the pre-computed DF22 controller. The function is coded in FPU32 assembly.

Parameters:

p	The DCL_DF22 structure
ek	The servo error
uk	The control effort in the previous sample interval

Return: The control effort

DCL_runDF22_C4 — Run the DF22 Full Compensator

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runDF22_C4(DCL_DF22 *p, float32_t ek)

Description: This function computes a full second order control law using the Direct Form 2 structure, and is identical in structure and operation to the C1 form. The function is coded in inline C.

Parameters:

p	The DCL_DF22 structure
ek	The servo error

Return: The control effort

DCL_runDF22_C5 — Run the Immediate DF22 Compensator

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runDF22_C5(DCL_DF22 *p, float32_t ek)

Description: This function computes the immediate part of the pre-computed DF22 controller. The function is identical in structure and operation to the C2 form. The function is coded in inline C.

Parameters:

p	The DCL_DF22 structure
ek	The servo error

Return: The control effort

DCL_runDF22_C6 — Run the Partial DF22 Compensator

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runDF22_C6(DCL_DF22 *p, float32_t ek, float32_t uk)

Description: This function computes the partial result of the pre-computed DF22 controller. The function is identical in structure and operation to the C3 form. The function is coded in inline C.

Parameters:

p	The DCL_DF22 structure
ek	The servo error
uk	The control effort in the previous sample interval

Return: The control effort

DCL_runDF22_L1 — Run the DF22 Full Compensator on the CLA

Header File: DCLCLA.h

Source File: DCL_DF22_L1.asm

Declaration: float32_t DCL_runDF22_L1(DCL_DF22_CLA *p, float32_t ek)

Description: This function computes a full third order control law using the Direct Form 2 structure, and is identical in structure and operation to the C1 form. The function is coded in CLA assembly language.

Parameters:

p	The DCL_DF22_CLA structure
ek	The servo error

Return: The control effort

DCL_runDF22_L2 — ***Run the Immediate DF22 Compensator on the CLA***

Header File: DCLCLA.h

Source File: DCL_DF22_L2L3.asm

Declaration: float32_t DCL_runDF22_L2(DCL_DF22_CLA *p, float32_t ek)

Description: This function computes the immediate part of the pre-computed DF22 controller. The function is identical in structure and operation to the C2 form. The function is coded in CLA assembly language.

Parameters:

p	The DCL_DF22_CLA structure
ek	The servo error

Return: The control effort

DCL_runDF22_L3 — ***Run the Partial DF22 Compensator on the CLA***

Header File: DCLCLA.h

Source File: DCL_DF22_L2L3.asm

Declaration: float32_t DCL_runDF22_L3(DCL_DF22_CLA *p, float32_t ek, float32_t uk)

Description: This function computes the partial result of the pre-computed DF22 controller. The function is identical in structure and operation to the C3 form. The function is coded in CLA assembly language.

Parameters:

p	The DCL_DF22_CLA structure
ek	The servo error
uk	The control effort in the previous sample interval

Return: The control effort

DCL_runDF22_L4 — ***Run the DF22 Full Compensator on the CLA***

Header File: DCLCLA.h

Source File: N/A

Declaration: float32_t DCL_runDF22_L4(DCL_DF22_CLA *p, float32_t ek)

Description: This function computes a full third order control law using the Direct Form 2 structure, and is identical in structure and operation to the C1 form. The function is coded in C.

Parameters:

p	The DCL_DF22_CLA structure
ek	The servo error

Return: The control effort

DCL_resetDF22 — Resets the DF22 Compensator

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_resetDF22(DCL_DF22 *p)

Description: This function resets the internal variables in the DCL_DF22 structure to default values. The forward and return path coefficients are configured to implement a unity gain response, and the `err` field in the CSS sub-structure is set NONE. Note that the function is atomic.

Parameters: p The DCL_DF22 structure

Return: Void

DCL_updateDF22 — Updates the DF22 Compensator Parameters

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_updateDF22(DCL_DF22 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the DF22 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_DF22 structure

Return: Void

DCL_fupdateDF22 — Updates the DF22 Controller Parameters

Header File: DCLF32.h

Source File: DCL_futils.asm

Declaration: void DCL_fupdateDF22(DCL_DF22 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the DF22 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters. The function is implemented as an assembly module.

Parameters: p The DCL_DF22 structure

Return: Void

DCL_isStableDF22 — Determines whether the DF22 Compensator is Stable

Header File: DCLF32.h

Source File: N/A

Declaration: int16_t DCL_isStableDF22(DCL_DF22 *p)

Description: This function determines whether the coefficient set in the SPS sub-structure represent a stable compensator. If the pole magnitude is less than one, the function returns '1', indicating stability; otherwise the function returns '0'. Refer to section 1.3.7 for more information on compensator stability tests.

Parameters: p The DCL_DF22 structure

Return: '1' if stable, otherwise '0'

DCL_loadDF22asZPK — Loads the DF22 Compensator from ZPK

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadDF22asZPK (DCL_DF22 *p, DCL_ZPK3 *q)

Description: This function loads the DF22 compensator coefficients in the SPS sub-structure from a 2-pole, 2-zero description held in a ZPK3 structure. Active coefficients are unaffected until the `DCL_updateDF22()` function is called. Refer to section 1.3.6 for more information on the ZPK3 structure.

Parameters: p The DCL_DF22 structure

q The DCL_ZPK3 structure

Return: Void

DCL_loadDF22asParallelPID — Loads the DF22 Compensator from a PID Description

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadDF22asParallelPID (DCL_22 *p, float32_t Kp, float32_t Ki, float32_t Kd, float32_t fc)

Description: This function loads the DF22 compensator coefficients in the SPS sub-structure to emulate a parallel form PID controller. Active coefficients are unaffected until the `DCL_updateDF22()` function is called.

Parameters:	p	The DCL_DF22 structure
	Kp	The equivalent parallel form PID controller proportional gain
	Ki	The equivalent parallel form PID controller integral gain
	Kd	The equivalent parallel form PID controller derivative gain
	fc	The equivalent derivative filter bandwidth in Hz
Return:	Void	

DCL_loadDF22asSeriesPID	***Loads the DF22 Compensator from a PID Description***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadDF22asSeriesPID (DCL_22 *p, float32_t Kp, float32_t Ki, float32_t Kd, float32_t fc)

Description: This function loads the DF22 compensator coefficients in the SPS sub-structure to emulate a series form PID controller. Active coefficients are unaffected until the `DCL_updateDF22()` function is called.

Parameters:	p	The DCL_DF22 structure
	Kp	The equivalent series form PID controller proportional gain
	Ki	The equivalent series form PID controller integral gain
	Kd	The equivalent series form PID controller derivative gain
	fc	The equivalent derivative filter bandwidth in Hz
Return:	Void	

3.9 Direct Form 2 (Third Order) Compensators

3.9.1 Description

The third order Direct Form 2 compensator (DF23) is similar in all respects to the DF22 compensator. Separate full and pre-computed forms are supplied in C and assembly for computation on the FPU32, and in assembly for computation on the CLA.

The control law is the same as the DF13 compensator.

$$u(k) = b_0 e(k) + b_1 e(k-1) + b_2 e(k-2) + b_3 e(k-3)$$
$$-a_1 u(k-1) - a_2 u(k-2) - a_3 u(k-3)$$

Equation 22.

A diagrammatic representation of the full third order Direct Form 2 compensator is shown below:

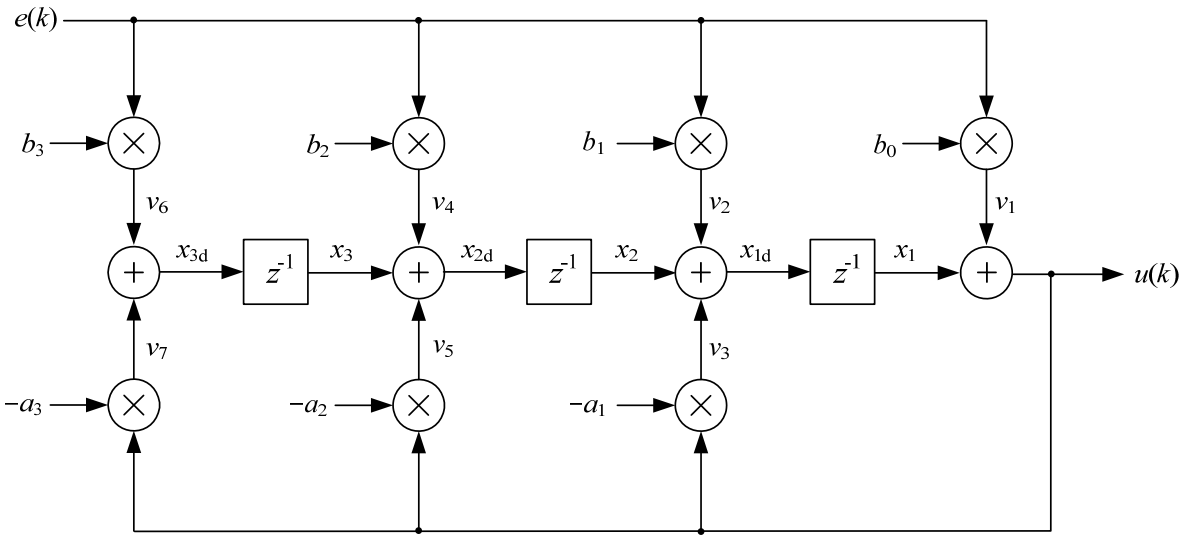


Figure 27. DCL_DF23 C1, C4, & L1 architecture

Sample-to-output delay can be reduced through the use of pre-computation, in a similar way to the DF22 compensator. In the k[th] interval, the immediate part is computed.

Equation 23. $u(k) = b_0 e(k) + v(k)$

Next, the v(k) partial result is pre-computed for use in the (k+1)[th] interval.

Equation 24. $v(k+1) = b_1 e(k) + b_2 e(k-1) + b_3 e(k-2) - a_1 u(k) - a_2 u(k-1) - a_3 u(k-2)$

The pre-computed form of DF23 is shown in the following diagrams:

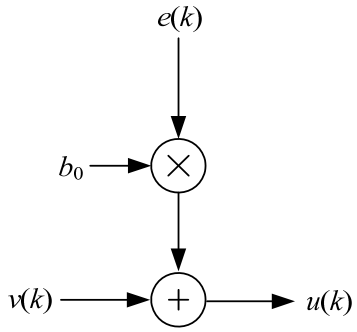


Figure 28. DCL_DF23 C2, C5, & L2 architecture

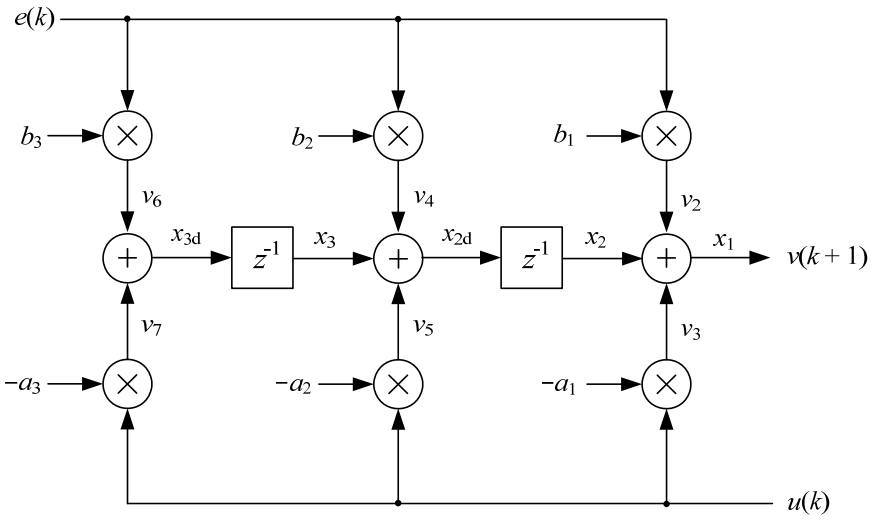


Figure 29. DCL_DF23 C3, C6, & L3 architecture

3.9.2 Implementation

All DF23 functions use a common C structure to hold coefficients and data, defined in the header file `DCL.h`. and `DCLCLA.h`.

It is the responsibility of the user to initialize both arrays prior to use. A set of default values is defined in the library header file and can be used with the variable declaration. An example of an initialized DCL_DF23 structure declaration is shown below:

```
DCL_DF23 myCtrl = DF23_DEFAULTS;
```

3.9.3 Functions

DCL_runDF23_C1	***Run the DF23 Full Compensator***

Header File: DCLF32.h

Source File: DCL_DF23_C1.asm

Declaration: float32_t DCL_runDF23_C1(DCL_DF23 *p, float32_t ek)

Description: This function computes a full third order control law using the Direct Form 2 structure. The function is coded in FPU32 assembly.

Parameters:

p	The DCL_DF23 structure
ek	The servo error

Return: The control effort

DCL_runDF23_C2 — ***Run the Immediate DF23 Compensator***

Header File: DCLF32.h

Source File: DCL_DF23_C2C3.asm

Declaration: float32_t DCL_runDF23_C2(DCL_DF23 *p, float32_t ek)

Description: This function computes the immediate part of the pre-computed DF23 controller. The function is coded in FPU32 assembly.

Parameters:

p	The DCL_DF23 structure
ek	The servo error

Return: The control effort

DCL_runDF23_C3 — ***Run the Partial DF23 Compensator***

Header File: DCLF32.h

Source File: DCL_DF23_C2C3.asm

Declaration: void DCL_runDF23_C3(DCL_DF23 *p, float32_t ek, float32_t uk)

Description: This function computes the partial result of the pre-computed DF23 controller. The function is coded in FPU32 assembly.

Parameters:

p	The DCL_DF23 structure
ek	The servo error
uk	The control effort in the previous sample interval

Return: The control effort

DCL_runDF23_C4 — ***Run the DF23 Full Compensator***

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runDF23_C4(DCL_DF23 *p, float32_t ek)

Description: This function computes a full third order control law using the Direct Form 1 structure, and is identical in structure and operation to the C1 form. The function is coded in inline C.

Parameters:

p	The DCL_DF23 structure
ek	The servo error

Return: The control effort

DCL_runDF23_C5 — Run the Immediate DF23 Compensator

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runDF23_C5(DCL_DF23 *p, float32_t ek)

Description: This function computes the immediate part of the pre-computed DF23 controller. The function is identical in structure and operation to the C2 form. The function is coded in inline C.

Parameters:

p	The DCL_DF23 structure
ek	The servo error

Return: The control effort

DCL_runDF23_C6 — Run the Partial DF23 Compensator

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runDF23_C6(DCL_DF23 *p, float32_t ek, float32_t uk)

Description: This function computes the partial result of the pre-computed DF23 controller. The function is identical in structure and operation to the C3 form. The function is coded in inline C.

Parameters:

p	The DCL_DF23 structure
ek	The servo error
uk	The control effort in the previous sample interval

Return: The control effort

DCL_runDF23_L1 — Run the DF23 Full Compensator on the CLA

Header File: DCLCLA.h

Source File: DCL_DF23_L1.asm

Declaration: float32_t DCL_runDF23_L1(DCL_DF23_CLA *p, float32_t ek)

Description: This function computes a full third order control law using the Direct Form 2 structure, and is identical in structure and operation to the C1 form. The function is coded in CLA assembly language.

Parameters:

p	The DCL_DF23_CLA structure
ek	The servo error

Return: The control effort

DCL_runDF23_L2 — ***Run the Immediate DF23 Compensator on the CLA***

Header File: DCLCLA.h

Source File: DCL_DF23_L2L3.asm

Declaration: float32_t DCL_runDF23_L2(DCL_DF23_CLA *p, float32_t ek)

Description: This function computes the immediate part of the pre-computed DF23 controller. The function is identical in structure and operation to the C2 form. The function is coded in CLA assembly language.

Parameters:

p	The DCL_DF23_CLA structure
ek	The servo error

Return: The control effort

DCL_runDF23_L3 — ***Run the Partial DF23 Compensator on the CLA***

Header File: DCLCLA.h

Source File: DCL_DF23_L2L3.asm

Declaration: float32_t DCL_runDF23_L3(DCL_DF23_CLA *p, float32_t ek, float32_t uk)

Description: This function computes the partial result of the pre-computed DF23 controller. The function is identical in structure and operation to the C3 form. The function is coded in CLA assembly language.

Parameters:

p	The DCL_DF23_CLA structure
ek	The servo error
uk	The control effort in the previous sample interval

Return: The control effort

DCL_resetDF23 — ***Resets the DF23 Compensator***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_resetDF23(DCL_DF23 *p)

Description: This function resets the internal variables in the DCL_DF23 structure to default values. The forward and return path coefficients are configured to implement a unity gain response, and the `err` field in the CSS sub-structure is set NONE. Note that the function is atomic.

Parameters:

p	The DCL_DF23 structure

Return: Void

DCL_updateDF23 — Updates the DF23 Compensator Parameters

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_updateDF23(DCL_DF23 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the DF23 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_DF23 structure

Return: Void

DCL_fupdateDF23 — Updates the DF23 Controller Parameters

Header File: DCLF32.h

Source File: DCL_futils.asm

Declaration: void DCL_fupdateDF23(DCL_DF23 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the DF23 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters. The function is implemented as an assembly module.

Parameters: p The DCL_DF23 structure

Return: Void

DCL_isStableDF23 — Determines whether the DF23 Compensator is Stable

Header File: DCLF32.h

Source File: N/A

Declaration: int16_t DCL_isStableDF23(DCL_DF23 *p)

Description: This function determines whether the coefficient set in the SPS sub-structure represent a stable compensator. If the pole magnitude is less than one, the function returns ‘1’, indicating stability; otherwise the function returns ‘0’. Refer to section 1.3.7 for more information on compensator stability tests.

Parameters: p The DCL_DF23 structure

Return: '1' if stable, otherwise '0'

DCL_loadDF23asZPK	Loads the DF23 Compensator from ZPK

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadDF23asZPK(DCL_DF23 *p, DCL_ZPK3 *q)

Description: This function loads the DF23 compensator coefficients in the SPS sub-structure from a 3-pole, 3-zero description held in a ZPK3 structure. Active coefficients are unaffected until the `DCL_updateDF23()` function is called. Refer to section 1.3.6 for more information on the ZPK3 structure.

Parameters: p The DCL_DF23 structure

q The DCL_ZPK3 structure

Return: Void

3.10 Fixed-Point PID Controllers

3.10.1 Description

The DCL contains one implementation of a parallel form fixed-point PID controller. The structure is similar to the floating-point C1 controller. Refer to section 3.1 for more information.

3.10.2 Implementation

The linear PID controller in the DCL32 includes the following features.

- Parallel form
- Programmable output saturation
- Anti-windup integrator reset
- Programmable low-pass derivative filter
- Feedback input to derivative path

Both PID and PI type controllers in the DCl32 library implement integrator anti-windup reset in a similar way. A clamp is present at the controller output which allows the user to set upper and lower limits on the control effort. If either limit is exceeded, an internal floating-point controller variable changes from logical 1 to logical 0. This variable is converted into Q24 format and multiplied by the integrator input, such that the integrator accumulates successive zero data when the output is saturated, avoiding the "wind-up" phenomenon.

The following equations describe the implementation of the PID32 controller. Note that the storage of static variables $\{i_{14}\ i_{10}\ d_2\ d_3\}$ is not shown.

The servo error equation is

Equation 25. $v_5(k) = r(k) - y(k)$

The proportional path equation is

Equation 26. $v_6(k) = K_p v_5(k)$

The integral path equations are

Equation 27. $v_7(k) = K_i v_5(k)$

Equation 28. $v_{14}(k) = v_{12}(k-1) v_7(k)$

Equation 29. $v_8(k) = v_{14}(k) + v_8(k-1)$

The derivative path equations are

Equation 30. $v_1(k) = K_d v_5(k)$

Equation 31. $v_2(k) = c_1 v_1(k)$

Equation 32. $v_4(k) = v_2(k) - v_2(k-1) - v_3(k-1)$

Equation 33. $v_3(k) = c_2 v_4(k)$

Note that the derivative coefficient c_1 must be divided by two on initialization. This element is typically much larger than c_2 so we enter half its value in the code and multiply twice. This allows greater numerical range for a given Q-format.

The output path equations are

Equation 34. $v_9(k) = v_6(k) + v_8(k) + v_4(k)$

Equation 35. $$u(k) = \begin{cases} v_9(k) & : u_{\min} < v_9(k) < u_{\max} \\ u_{\max} & : v_9(k) \geq u_{\max} \\ u_{\min} & : v_9(k) \leq u_{\min} \end{cases}$$

Equation 36. $v_{11}(k) = u(k) - v_9(k)$

Equation 37. $$v_{12}(k) = \begin{cases} 1 & : v_{11}(k) = 0 \\ 0 & : v_{11}(k) \neq 0 \end{cases}$$

Equation 38. $i_{14}(k) = v_{12}(k)$

The DCL_PID32 implementation is shown below:

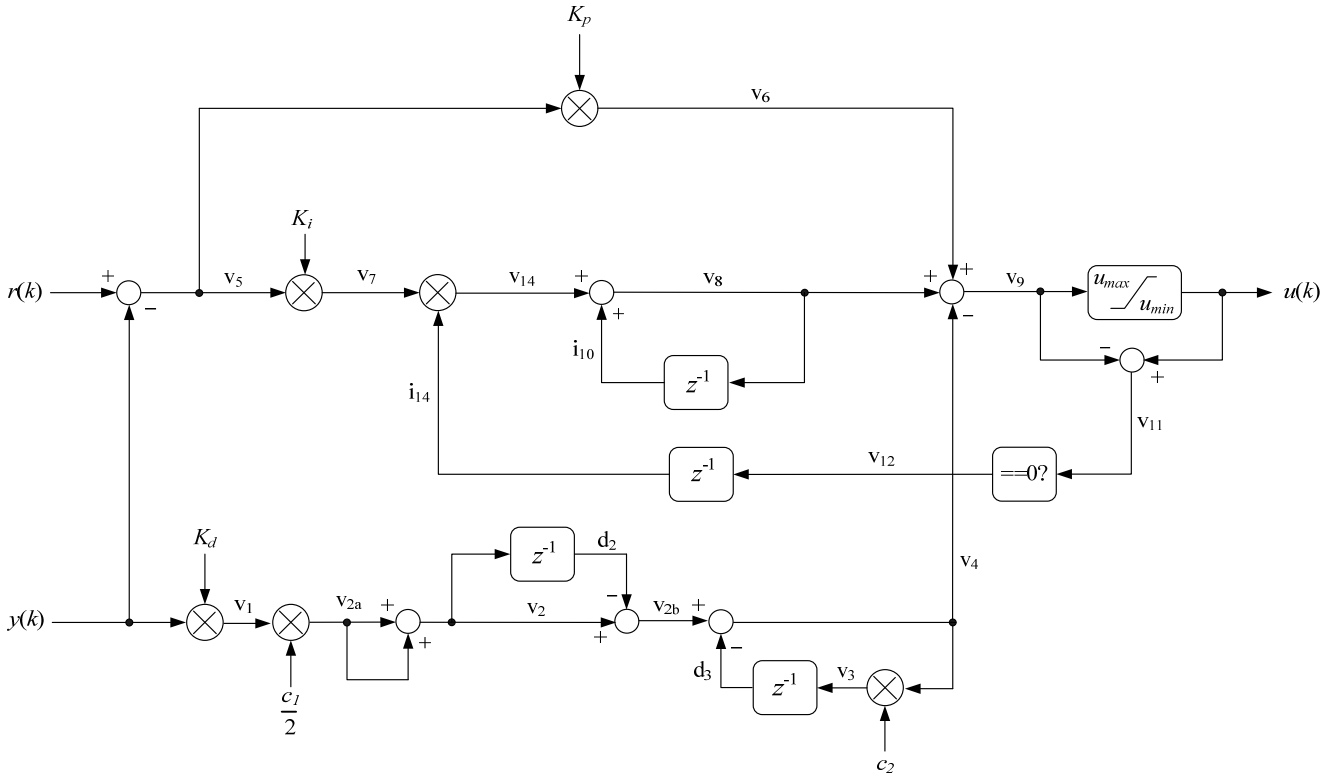


Figure 30. DCL_PID32 A1 architecture

The linear PID_A1 controller uses a C structure to hold coefficients and data, defined in the header file `DCLC28.h`. The order of these structure elements must not be changed by the user.

3.10.3 Functions

DCL_runPID_A1	***Run the Parallel Form PID32 Controller***

Header File:	DCLC28.h	
Source File:	DCL_PID_A1.asm	
Declaration:	int32_t DCL_runPID_A1(DCL_PID32 *p, int32_t rk, int32_t yk)	
Description:	This function executes a parallel form PID controller on the C28x. The function is coded in C28x assembly. All input and output variables are in Q24 format.	
Parameters:	p	The PID32 structure
	rk	The controller set-point reference
	yk	The measured feedback
Return:	The control effort	

DCL_resetPID32	***Resets the PID Controller***

Header File: DCLC28.h

Source File: N/A

Declaration: void DCL_resetPID32(DCL_PID32 *p)

Description: This function resets the internal variables in the DCL_PID32 structure to default values. The integrator accumulator and store derivative path values are set to zero, and the integrator clamp variable set to one. The function also sets the `err` field in the CSS sub-structure is set NONE. Note that the function is atomic.

Parameters: p The DCL_PID32 structure

Return: Void

DCL_updatePID32	***Updates the PID32 Controller Parameters***

Header File: DCLC28.h

Source File: N/A

Declaration: void DCL_updatePID32(DCL_PID32 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PID structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_PID32 structure

Return: Void

DCL_fupdatePID32	***Updates the PID32 Controller Parameters***

Header File: DCLC28.h

Source File: DCL_futils32.asm

Declaration: void DCL_fupdatePID32(DCL_PID32 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PID32 structure and `sts` is cleared. This function is implemented as an assembly module and does not perform any error checking. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_PID32 structure

Return: Void

3.11 Fixed-Point PI Controllers

3.11.1 Description

The DCL contains one implementation of a fixed-point series form PI controller. The PI is similar in operation to the PID controller, with the removal of the derivative path. Refer to section 3.2 for more information.

3.11.2 Implementation

The following equations describe the implementation of the PI32 controller.

Equation 39. $v_1(k) = r(k) - y(k)$

Equation 40. $v_2(k) = K_p v_5(k)$

Equation 41. $v_3(k) = K_i v_2(k)$

Equation 42. $v_8(k) = v_9(k-1) v_3(k)$

Equation 43. $v_4(k) = v_8(k) + v_4(k-1)$

Equation 44. $v_5(k) = v_2(k) + v_4(k)$

Equation 45.
$$u(k) = \begin{cases} v_5(k) & : u_{\min} < v_5(k) < u_{\max} \\ u_{\max} & : v_5(k) \geq u_{\max} \\ u_{\min} & : v_5(k) \leq u_{\min} \end{cases}$$

Equation 46. $v_7(k) = u(k) - v_5(k)$

Equation 47.
$$v_9(k) = \begin{cases} 1 & : v_7(k) = 0 \\ 0 & : v_7(k) \neq 0 \end{cases}$$

The ideal form PI implementation is shown below:

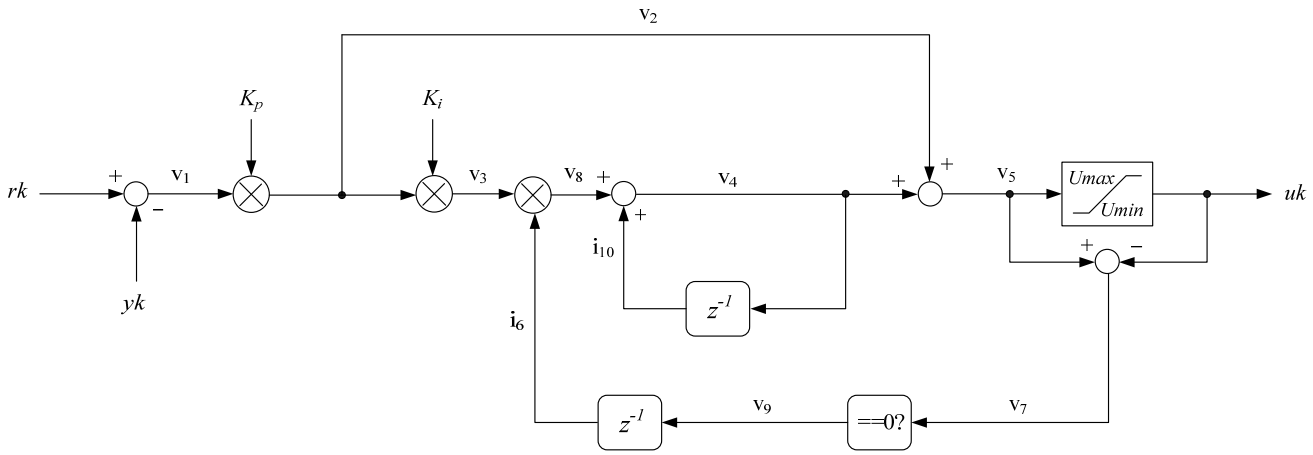


Figure 31. DCL_PI A1 architecture

The linear PI controller uses a C structure to hold coefficients and data, defined in the header file `DCLC28.h`. The order of these structure elements must not be changed by the user.

3.11.3 Functions

DCL_runPI_A1	***Run the Ideal Form PI32 Controller***

Header File: DCLC28.h

Source File: DCL_PI_A1.asm

Declaration: int32_t DCL_runPI_A1(DCL_PI32 *p, int32_t rk, int32_t yk)

Description: This function executes an ideal form PI32 controller on the C28x. The function is coded in C28x assembly. All input and output variables are in Q24 format.

Parameters:

p	The PI32 structure
rk	The controller set-point reference
yk	The measured feedback

Return: The control effort

DCL_resetPI32	***Resets the PI Controller***

Header File: DCLC28.h

Source File: N/A

Declaration: void DCL_resetPI32(DCL_PI32 *p)

Description: This function resets the internal variables in the DCL_PI32 structure to default values. The integrator accumulator and store derivative path values are set to

zero, and the integrator clamp variable set to one. The function also sets the `err` field in the CSS sub-structure is set NONE. Note that the function is atomic.

Parameters: p The DCL_PI32 structure

Return: Void

DCL_updatePI32 — ***Updates the PI32 Controller Parameters***

Header File: DCLC28.h

Source File: N/A

Declaration: void DCL_updatePI32(DCL_PI32 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PID structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_PI32 structure

Return: Void

DCL_fupdatePI32 — ***Updates the PI32 Controller Parameters***

Header File: DCLC28.h

Source File: DCL_futils32.asm

Declaration: void DCL_fupdatePI32(DCL_PI32 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PI32 structure and `sts` is cleared. This function is implemented as an assembly module and does not perform any error checking. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_PI32 structure

Return: Void

3.12 Gain Scheduler Module

3.12.1 Description

The DCL contains an implementation of a basic Gain Scheduler Module (GSM) which runs on the FPU32. The GSM works by dividing the positive normalized input range, from 0 to +1, into eight equal sectors, each of which is associated with a separate gain. The negative input range, from 0 to -1, has the same gains but with a sign change.

As the input sweeps through the full range from -1 to +1, the output changes in a way determined by the entries in two look-up tables loaded by the user. One table fixes the gain in each sector, while the other fixes the offsets at the sector boundaries. In this way, the user may realize a piecewise continuous non-linear input-output function without introducing step discontinuities into the control.

3.12.2 Implementation

The sector gain look-up table consists of eight entries covering the normalized positive input range. The sector offset table consists of nine entries, with the first entry set to zero, and the final entry defining the output when the input is equal to 1. Figure 32 shows the sector numbering used in the GSM for a typical target curve. Sectors, gains, and offsets are denoted *S*, *m*, and *c* respectively. Note the symmetry for positive and negative inputs, and the clamp characteristic on the upper right when the input magnitude exceeds 1.

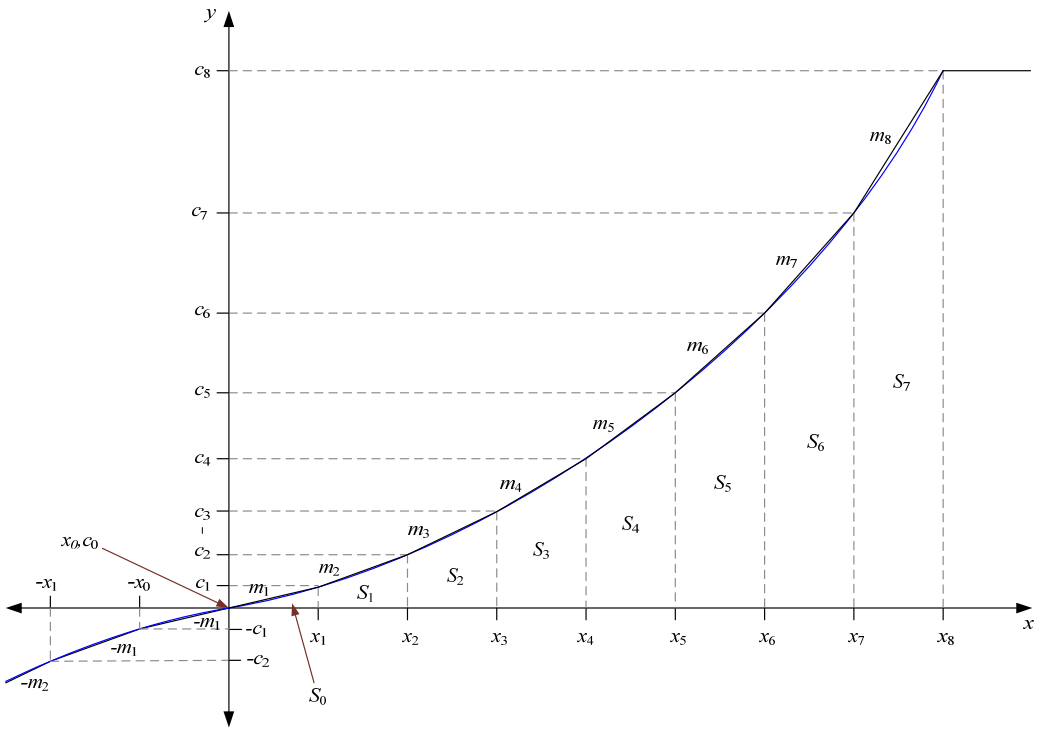


Figure 32. *DCL_GSM sector numbering*

A typical scenario is that the user will identify a target function from which the GSM curve will be loaded. The user will load each of the nine offsets in the SPS sub-structure from the target function, then call the function `DCL_loadGSMgains()` to initialize the gain

table. Once this is done, the SPS tables can be copied into the active parameter set using the `DCL_updateGSM()` function. Refer to code example 7 in chapter 5 to see how this might be done.

The Matlab script file `GSM_example.m` can be found in the \models sub-directory and demonstrates how the gain and offset tables are initialized. Note that due to array indexing differences between Matlab and C, sector initialization is slightly different to the DCL code.

3.12.3 Functions

DCL_runGSM_C1	***Run the Gain Scheduler Module***

Header File: DCLF32.h

Source File: N/A

Declaration: float32_t DCL_runGSM_C1(DCL_GSM *p, float32_t x)

Description: This function runs the gain scheduler to determine an output from the gain and offset arrays in the DCL_GSM structure. The function is coded in in-line C.

Parameters: p The DCL_GSM structure

x The normalized input

Return: The gain adjusted output

DCL_resetGSM	***Resets the GSM Module***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_resetGSM(DCL_GSM *p)

Description: This function resets the internal variables in the DCL_GSM structure to default values. All gain segments are set to unity gain and the offset array configured to generate a linear input/output relationship. The function also sets the `err` field in the CSS sub-structure set NONE. Note that the function is atomic.

Parameters: p The DCL_GSM structure

Return: Void

DCL_updateGSM	***Updates the GSM Parameters***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_updateGSM(DCL_GSM *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the active DCL_GSM structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_GSM structure

Return: Void

DCL_fupdateGSM — ***Updates the GSM Controller Parameters***

Header File: DCLF32.h

Source File: DCL_futils.asm

Declaration: void DCL_fupdateGSM(DCL_GSM *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the GSM structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters. The function is implemented as an assembly module.

Parameters: p The DCL_GSM structure

Return: Void

DCL_loadGSMoffsets — ***Loads the GSM Offset Parameters***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadGSMoffsets(DCL_GSM *p)

Description: This function configures the offset array in the SPS sub-structure to produce a piecewise continuous input-output curve from the gains. The active parameters are not affected until the `DCL_updateGSM()` function is called.

Parameters: p The DCL_GSM structure

Return: Void

DCL_loadGSMgains — ***Loads the GSM Gain Parameters***

Header File: DCLF32.h

Source File: N/A

Declaration: void DCL_loadGSMgains(DCL_GSM *p)

Description: This function configures the gain array in the SPS sub-structure to produce a piecewise continuous input-output curve from the offsets. The active parameters are not affected until the `DCL_updateGSM()` function is called.

Parameters: p The DCL_GSM structure

Return: Void

3.13 Non-linear Control Law

3.13.1 Description

The DCL contains an implementation of the non-linear control law used in the NLPID & NLPI controller described earlier in this section. The user could apply this function to implement a gain scheduling type of control.

3.13.2 Implementation

Refer to section 3.3.1 for information on the non-linear law.

3.13.3 Functions

DCL_runNLF_C1	***Run the NLF Control Law***

Header File: DCL_NLPID.h

Source File: N/A

Declaration: float32_t DCL_runNLF_C1(float32_t x, float32_t alpha, float32_t delta)

Description: This function executes a non-linear control law defined by the parameters in the DCL_NLF structure. No error checks are performed on the parameters. The function is coded in inline C.

Parameters: x The input variable

alpha The non-linear exponent

delta The linear region semi-width

Return: The non-linear result

3.14 Double Precision PID Controllers

3.14.1 Description

The DCL contains one implementation of a linear PID controller in double precision floating point form. This controller may be used with the FPU32 CPU, however support for the double precision data type currently relies on the run time support libraries which are not cycle efficient. The structure of the controller is identical to the PID_C1 & PID_C2 controllers described earlier in this chapter.

Support functions for PIDF64 do not currently include the ability to load the controller from transfer function coefficient or ZPK3 descriptions.

3.14.2 Implementation

The controller is supplied in inline C source. Refer to section 3.1.2 for further information.

3.14.3 Functions

DCL_runPIDF64_S1	***Run the PIDF64 Controller***

Header File: DCLF64.h

Source File: N/A

Declaration: float64_t DCL_runPIDF64_S1(DCL_PIDF64 *p, float64_t rk, float64_t yk, float32_t lk)

Description: This function executes an ideal form PIDF64 controller on the FPU32, and is identical in structure and operation to the C1 & C2 forms. The function is coded in inline C.

Parameters:

p	The DCL_PIDF64 structure
rk	The controller set-point reference
yk	The measured feedback
lk	External output clamp flag

Return: The control effort

DCL_resetPIDF64	***Resets the PIDF64 Controller***

Header File: DCLF64.h

Source File: N/A

Declaration: void DCL_resetPIDF64(DCL_PIDF64 *p)

Description: This function resets the internal variables in the DCL_PIDF64 structure to default values. The integrator accumulator and store derivative path values are set to 0.0, and the integrator clamp variable set to 1.0. The function also sets the `err` field in the CSS sub-structure to NONE. Note that the function is atomic.

Parameters: p The DCL_PIDF64 structure

Return: Void

DCL_updatePIDF64 — ***Updates the PIDF64 Controller Parameters***

Header File: DCLF64.h

Source File: N/A

Declaration: void DCL_updatePIDF64(DCL_PIDF64 *p)

Description: This function tests the `sts` field in the CSS sub-structure to determine whether a parameter update is required. If so, the parameters stored in the SPS sub-structure are copied into the PIDF64 structure and `sts` is cleared. Note that the function is atomic. Refer to section 2.4 for more information on updating controller parameters.

Parameters: p The DCL_PIDF64 structure

Return: Void

DCL_setPIDF64filterBW — ***Set the PIDF64 Derivative Filter Bandwidth***

Header File: DCLF64.h

Source File: N/A

Declaration: void DCL_setPIDF64filterBW(DCL_PIDF64 *p, float64_t fc)

Description: Loads the derivative filter coefficients `c1` & `c2` in the SPS based on the desired filter bandwidth specified in Hz. Coefficients in the active parameter set are unaffected until the controller is updated using DCL_updatePIDF64().

Parameters: p The DCL_PIDF64 structure

fc The desired filter bandwidth in Hz

Return: Void

DCL_setActivePIDF64filterBW — ***Set the Active PIDF64 Derivative Filter Bandwidth***

Header File: DCLF64.h

Source File: N/A

Declaration: void DCL_setActivePIDF64filterBW(DCL_PIDF64 *p, float64_t fc, float64_t T)

Description: Loads the derivative filter coefficients `c1` & `c2` in the active PIDF64 structure based on the desired filter bandwidth specified in Hz and the controller update date in seconds. This function does not use or modify the SPS.

Parameters:

p	The DCL_PIDF64 structure
fc	The desired filter bandwidth in Hz
T	The controller update rate in seconds

Return: Void

DCL_getPIDF64filterBW — ***Get the PIDF64 Derivative Filter Bandwidth***

Header File: DCLF64.h

Source File: N/A

Declaration: float64_t DCL_getPIDF64filterBW(DCL_PIDF64 *p)

Description: Finds the bandwidth of the current derivative filter in Hz by examining the coefficients in the active parameter set (i.e. not the SPS).

Parameters:

p	The DCL_PIDF64 structure

Return: The active derivative filter bandwidth in Hz

Chapter 4

Utilities

This chapter describes the supporting functions included in the Digital Control Library.

Section

4.1 Control Clamps

4.2 Floating Point Data Logging Functions

4.3 4-channel Floating Point Data Logger

4.4 Transient Capture Module

4.5 Performance Measurement

4.6 Fixed Point Data Logging Functions

The Digital Controller Library includes a small number of utilities intended to support use of the library. These include:

- Clamp functions for the CPU and CLA
- Floating point data logging functions
- A 4-channel floating point data logger
- A Transient Capture Module
- Functions for measurement of control performance
- Fixed point data logging functions
- A 4-channel fixed point data logger

4.1 Control Clamps

4.1.1 Description

The library contains three functions for clamping a control variable to specified upper and lower limits. These would typically be used to impose a pre-defined bound the output of a controller function to prevent actuator saturation or overload. Saturation in a control loop must be handled with care since control of the system is effectively lost. Furthermore, controllers which implement integration of historical servo data can exhibit a phenomenon known as “wind-up”, in which the controller output increases in magnitude while the loop

is saturated. This condition leads to delay in recovering from the saturation because the accumulated controller output must be removed before it comes within range of the actuator and the controller resumes proper operation. For more information on integrator wind-up, refer to section 3.1.1 and ref. [3].

The clamp functions bound the input data variable to pre-determined limits and return a logical value 1 if either bound is matched or exceeded. If the input data lies definitely within limits (i.e. neither bound is matched or exceeded) the functions return logical 0. The return value can be used by PI & PID regulators to implement anti-windup reset, and may be used to clamp the output of the pre-computed forms of all direct form compensators. An example may be found in the DF22 example project supplied with the library (see chapter 5).

A difference exists between the C28x clamp function and that of the CLA. On the CPU the returned value is an unsigned integer of either 0 or 1, while the corresponding CLA function returns a floating point value of 0.0f or 1.0f. This is because the handling of fixed-point data on the CLA is less efficiently supported than on the main CPU.

4.1.2 Functions

DCL_runClamp_C1	***Floating-Point Data Clamp***

Header File: DCL.h

Source File: DCL_clamp_C1.asm

Declaration: uint16_t DCL_runClamp_C1(float *data, float Umax, float Umin)

Description: This function clamps a floating-point data value to defined limits and returns a non-zero integer if either limit is matched or exceeded. The function is coded in assembly.

Parameters:

data	The TCM structure
Umax	The upper data limit
Umin	The lower data limit

Return: 0 if the data lies definitely within limits, 1 if the data matches or exceeds the limits.

DCL_runClamp_C2	***Floating-Point Data Clamp***

Header File: DCL.h

Source File: N/A

Declaration: uint16_t DCL_runClamp_C2(float *data, float Umax, float Umin)

Description: This function clamps a floating-point data value to defined limits and returns a non-zero integer if either limit is matched or exceeded. The function is coded in inline C.

Parameters:

data	The TCM structure
Umax	The upper data limit

	Umin	The lower data limit

Return: 0 if the data lies definitely within the specified limits; 1 if the data matches or exceeds the specified limits.

DCL_runClamp_L1	***Floating-Point Data Clamp***

Header File: DCL.h

Source File: DCL_clamp_L1.asm

Declaration: float DCL_runClamp_L1(float *data, float Umax, float Umin)

Description: This function clamps a floating-point data value to defined limits and returns a non-zero floating-point result if either limit is matched or exceeded. The function is coded in CLA assembly.

Parameters:	data	The TCM structure
	Umax	The upper data limit
	Umin	The lower data limit

Return: 0.0f if the data lies definitely within the specified limits; 1.0f if the data matches or exceeds the specified limits.

4.2 Floating Point Data Logging Functions

4.2.1 Description

The Digital Control Library includes a general purpose floating-point data logger utility which is useful when testing and debugging control applications. The intended use of the data logger utility is to capture a stream of data values in a block of memory for subsequent analysis. The data logger is supplied in the form of a C header file and one assembly file, and it may be used on any C2000 device irrespective of whether the DCL is used. The utility may not be used on the CLA.

The data logger operates with arrays of 32-bit floating-point data. The location, size, and indexing of each array are defined by three pointers capturing the start address, end address, and data index address. All three pointers are held in a common C structure with the data type “FDLOG”, defined as follows:

```
typedef volatile struct {
      float *fptr;
      float *lptr;
      float *dptr;
} FDLOG;
```

Conceptually, the relationship between the array pointers and the elements of a data array of length “N” is shown below:

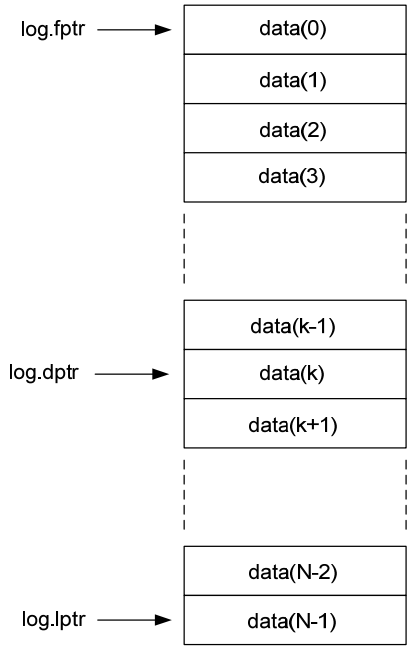


Figure 33. Data log pointer allocation

The data index pointer (dptr) always points to the next address to be written or read, and advances through the memory block as each new data value is written into the log. On reaching the end of the log, the pointer is reset to the first address in the log. The data logger header file contains a set of in-line C functions to access and manipulate data logs.

To use the data logger, you must include the header file `DCL_fdlog.h` in your project. Typically, a user would create an instance of an FDLOG structure as follows:

```
FDLOG myBuf = FDLOG_DEFAULTS;
```

The log pointers can then be initialized in the user's code such that they reference a memory block in a specific address range. Thereafter, the code can clear or load the buffer a specific data value, and then begin writing data into it using the `DCL_writeLog()` function. The DF22 example project shows how this is done.

The DCL also contains two functions which perform fast read and write to a data log. These are assembly coded functions in the source file `DCL_frwlog.asm`. The execution cycles for these and the corresponding C coded DCL functions are shown below:

Table 11. Data log read/write benchmarks

DCL_writeLog	48
DCL_readLog	39
DCL_fwriteLog	22
DCL_freadLog	22

4.2.2 Functions

DCL_deleteLog	***Delete a Data Log***

Header File: DCL_fdlog.h

Source File: N/A

Declaration: void DCL_deleteLog(FDLOG *p)

Description: This function resets all structure pointers to null value.

Parameters: p The FDLOG structure

Return: Void

DCL_resetLog	***Reset a Data Log***

Header File: DCL_fdlog.h

Source File: N/A

Declaration: void DCL_resetLog(FDLOG *p)

Description: This function resets the data index pointer to start of the data log.

Parameters: p The FDLOG structure

Return: Void

DCL_initLog	***Initialize a Data Log Structure***

Header File: DCL_fdlog.h

Source File: N/A

Declaration: void DCL_initLog(FDLOG *p, float32_t *addr, uint16_t size)

Description: This function assigns the buffer pointers to a memory block or array and sets the data index pointer to the first address.

Parameters: p The FDLOG structure

addr The start address of the memory block

size The length of the memory block in 32-bit words

Return: Void

DCL_writeLog — Write Data into a Log

Header File: DCL_fdlog.h

Source File: N/A

Declaration: float DCL_writeLog(FDLOG *p, float32_t data)

Description: This function writes a data point into the buffer and advances the indexing pointer, wrapping if necessary. The function returns the data value being over-written, which allows simple implementation of a fixed-length delay line.

Parameters:

p	The FDLOG structure
data	The input data value addr

Return: The over-written data value

DCL_fillLog — Fill a Data Log with Specified Data

Header File: DCL_fdlog.h

Source File: N/A

Declaration: void DCL_fillLog(FDLOG *p, float32_t data)

Description: This function fills the data log with a given data value and resets the data index pointer to the start of the log.

Parameters:

p	The FDLOG structure
data	The fill data value

Return: Void

DCL_clearLog — Fill a Data Log Contents with Zero

Header File: DCL_fdlog.h

Source File: N/A

Declaration: void DCL_clearLog(FDLOG *p)

Description: This function clears the buffer contents by writing 0 to all elements and resets the data index pointer to the start of the log.

Parameters:

p	The FDLOG structure

Return: Void

DCL_readLog — Fill a Data Log Contents with Zero

Header File: DCL_fdlog.h

Source File: N/A

Declaration: float32_t DCL_readLog(FDLOG *p)

Description: This function reads a data point from the buffer and then advanced the index pointer, wrapping if necessary.

Parameters: p The FDLOG structure

Return: The indexed data value

DCL_copyLog — Copies one Data Log into Another

Header File: DCL_fdlog.h

Source File: N/A

Declaration: void DCL_copyLog(FDLOG *p, FDLOG *q)

Description: This function copies the contents of one log into another and resets both buffer index pointers. The function assumes both logs have the same length.

Parameters: p The destination FDLOG structure

q The source FDLOG structure

Return: Void

DCL_freadLog — Performs Fast Read from a Data Log

Header File: DCL_fdlog.h

Source File: DCL_frwlog.asm

Declaration: float32_t DCL_freadLog(FDLOG *p)

Description: This function reads a data point from the log and then advances the indexing pointer, wrapping if necessary. This function is coded in assembly.

Parameters: p The FDLOG structure

Return: The indexed data value

DCL_fwriteLog — Performs Fast Write into a Data Log

Header File: DCL_fdlog.h

Source File: DCL_frwlog.asm

Declaration: float32_t DCL_fwriteLog(FDLOG *p, float32_t data)

Description: This function writes a data point into the buffer and advances the indexing pointer, wrapping if necessary. Returns the over-written data value for delay line or FIFO implementation. This function is coded in assembly.

Parameters: p The FDLOG structure

data The input data value

Return: The over-written data value

4.3 4-channel Floating Point Data Logger

4.3.1 Description

The Digital Control Library contains a 4 channel floating point data logger module denoted “MLOG”. This module uses the data logger functions described above to capture up to four channels of incoming data in separate buffers for later inspection. The MLOG module is trigged by a sample at its first input exceeding either of a pair of user defined thresholds, after which incoming samples are successively logged into each buffer until they are full. A useful feature of the MLOG module is that the sampling rate can be adjusted by the user to change the time scale of the capture frame.

Conceptually, the MLOG architecture consists of four floating point data capture frames, each of which is a buffer defined by an FDLOG structure. The input to each buffer passes through a sample scaler, which divides the sample rate by a user programmable integer. Sampling is initiated when data at the first input exceeds a pre-defined upper or lower limit, after which the four data samples are logged into the capture frames at the desired rate until the buffers are full.

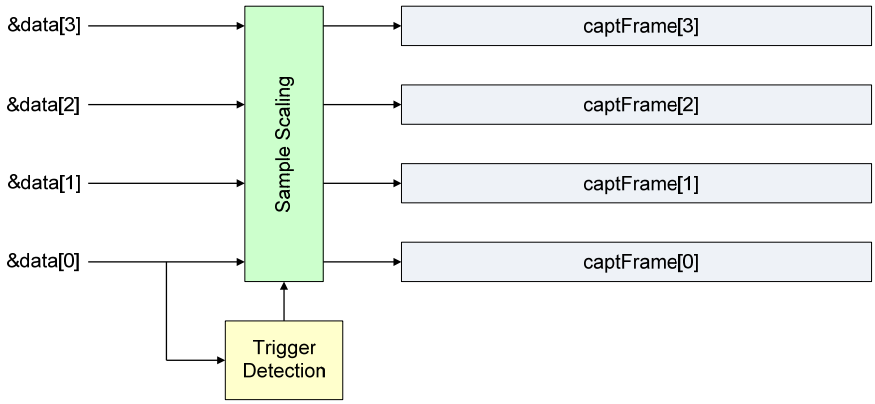


Figure 34. MLOG architecture

The MLOG module always operates in one of five modes:

- *MLOG_idle*
- *MLOG_armed*

- *MLOG_capture*
- *MLOG_complete*

The operating mode is stored in an element in the MLOG structure. All floating point MLOG functions are coded in inline C functions in the header file `MLOG.h`. Note that the DCL includes a separate fixed point 4-channel data logger which is described later in this chapter.

The MLOG module is defined by a C structure as follows:

```
typedef volatile struct {
      FDLOG captFrame[MLOG_CHANS];  //!< Capture data frames
      float32_t *data[MLOG_CHANS];  //!< Data channel pointers
      float32_t trigMax;            //!< Upper trigger threshold
      float32_t trigMin;            //!< Lower trigger threshold
      uint16_t tScale;              //!< Number of samples/write
      uint16_t sCount;              //!< Sample counter
      uint16_t mode;                //!< Operating mode
} MLOG;
```

4.3.2 Functions

DCL_initMLOG	***Initialize the MLOG Module***

Header File: DCL_MLOG.h

Source File: N/A

Declaration: void DCL_initMLOG(MLOG *q, float32_t *addr, uint16_t size, float32_t tmax, float32_t tmin, uint16_t div)

Description: This function loads all buffer contents with zero, and resets all buffer data pointers to their respective start addresses. The upper and lower trigger thresholds, and sample time scale are loaded into the MLOG structure. On completion, the MLOG module is in *MLOG_idle* mode.

Parameters:

q	The MLOG structure
addr	Pointer to the start address of the MLOG buffers
size	The size in 32-bit words of each buffer
tmax	The upper trigger threshold
tmin	The lower trigger threshold
div	The sample scaler

Return: Void

DCL_resetMLOG	***Reset the MLOG Module***

Header File: DCL_MLOG.h

Source File: N/A

Declaration: void DCL_resetMLOG(MLOG *q)

Description: This function loads all buffer contents with zero, and resets all buffer data pointers to their respective start addresses. On completion, the MLOG module is in *MLOG_idle* mode.

Parameters: q The MLOG structure

Return: Void

DCL_armMLOG — ***Arm the MLOG Module***

Header File: DCL_MLOG.h

Source File: N/A

Declaration: void DCL_armMLOG(MLOG *q)

Description: This function changes the MLOG operating mode from *MLOG_idle* to *MLOG_armed*. If the operating mode is not already *MLOG_idle* when the function is called, it sets the mode to *MLOG_idle*.

Parameters: q The MLOG structure

Return: Void

DCL_runMLOG — ***Run the MLOG Module***

Header File: DCL_MLOG.h

Source File: N/A

Declaration: void DCL_runMLOG(MLOG *p)

Description: This function runs the MLOG module. If in the *MLOG_armed* mode, the MLOG monitors its first input to determine whether either threshold has been exceeded. If so, the module enter *MLOG_capture* mode and sample collection begins. When the buffers are full, the operating mode is set to *MLOG_complete*.

Parameters: q The MLOG structure

Return: Void

4.4 Transient Capture Module

The Transient Capture Module (TCM) is a triggered data logger which captures a burst of incoming data. A typical use is the capture of a transient response following a step input to a control system. The trigger conditions are a pair of user defined limits on the incoming data. The capture process is triggered by the first data point which exceeds either limit.

A feature of the TCM is that it captures a programmable length lead frame, allowing the user to inspect conditions immediately prior to the trigger condition. This is accomplished with three FDLOG structures which are elements in the TCM data structure, together with the limit pair. Once initialized, the status of the TCM is captured in one of four enumerated operating modes:

- *TCM_idle*
- *TCM_armed*
- *TCM_capture*
- *TCM_complete*

The TCM data is contained in a C structure as shown below:

```
typedef volatile struct {
      FDLOG moniFrame;  //!< Monitor data frame
      FDLOG leadFrame;  //!< Lead data frame
      FDLOG captFrame;  //!< Capture data frame
      float trigMax;    //!< Upper trigger threshold
      float trigMin;    //!< Lower trigger threshold
      uint16_t mode;    //!< Operating mode
      uint16_t lead;    //!< Lead frame size in 32-bit words
} TCM;
```

The current mode is available in the mode element in the TCM structure. To use the TCM, the user must do the following:

1. Include the header file `TCM.h` in the project.
2. Allocate a RAM memory block to hold the full capture buffer.
3. Create an instance of the TCM structure and initialize it using `DCL_initTCM()`
4. Arm the TCM using `DCL_armTCM()`
5. Log data into the TCM using `DCL_runTCM()`
6. Monitor the mode element in the TCM structure to determine when the capture is complete.

A code example illustrating the use of the TCM is supplied with the library and is described in chapter 5.

In the following diagrams, lead, capture, and monitor frames are indexed using the FDLOG structures x, y, & z respectively (note that these are not the names used in the TCM structure). FDLOG pointers are color coded blue, green, and red, respectively. To help visualize the sequence of events, the diagram shows in light gray the data which will eventually be logged into the TCM, and in blue the current frame contents in each mode.

4.4.1 TCM_idle Mode

In *TCM_idle* mode the TCM buffers are as shown below. All buffer contents are zero, all frame data pointers are at the start of their respective frames and no data is being logged. This is the condition after the `DCL_initTCM()` function has been called.

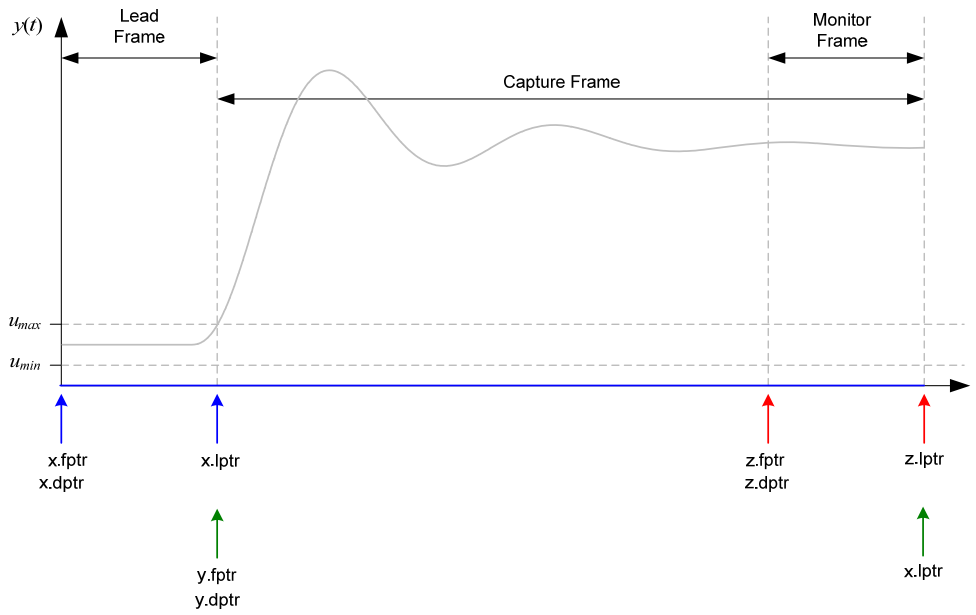


Figure 35. TCM operation in TCM_idle mode

4.4.2 TCM_armed Mode

The TCM is armed by a call to `DCL_armTCM()`. In this mode, incoming data is continually logged in the monitor frame. The monitor frame acts as a circular buffer, the index pointer wrapping to the start of the monitor frame when it reaches the end.

Each data point is compared with the upper and lower trigger thresholds to determine whether to initiate a capture sequence. As long as the incoming data remains within the specified limits, the TCM remains in *TCM_armed* mode.

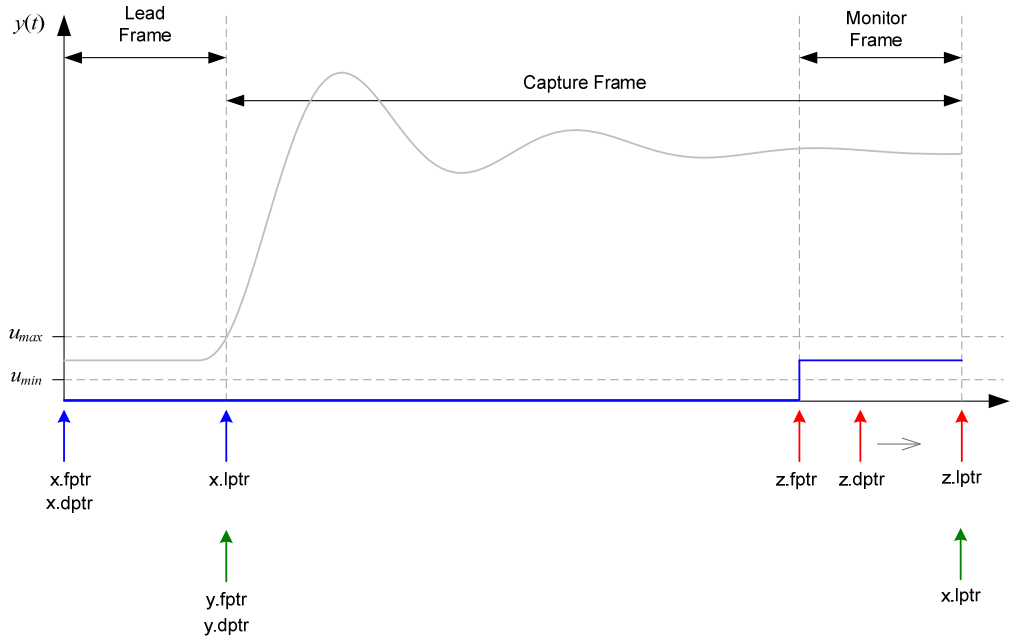


Figure 36. *TCM operation in TCM_armed mode*

4.4.3 TCM_capture Mode

The first data point which exceeds either trigger threshold initiates a capture sequence. The TCM automatically enters *TCM_capture* mode and incoming data is logged into the capture frame. Meanwhile, the monitor frame stops collecting data and starts to un-wind its contents into the lead frame. Notice that the monitor frame contains the lead data sequence, but the starting point is not aligned with the frame.

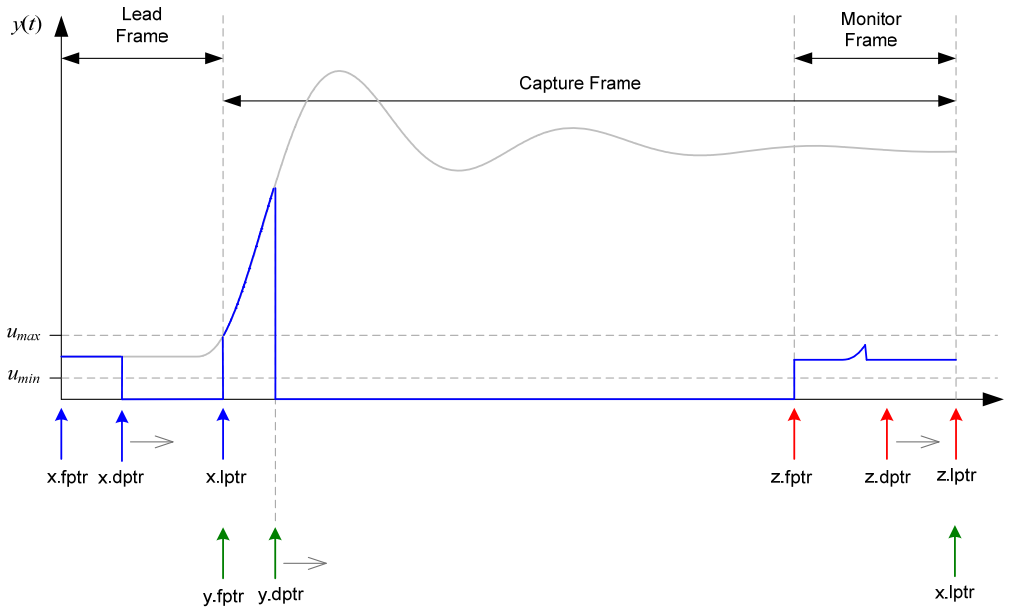


Figure 37. *TCM operation in capture mode (monitor frame un-winding)*

Once the lead frame is full, the monitor frame stops copying out its data. Incoming data continues being logged into the capture frame until it is full. The monitor frame contents have now been completely loaded into the lead frame and will be over-written.

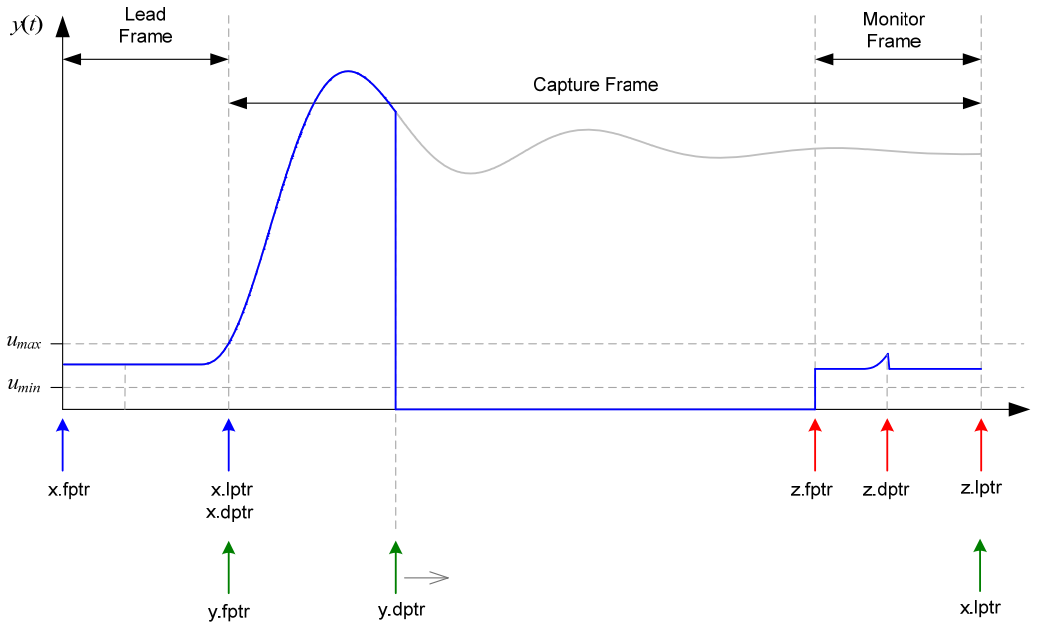


Figure 38. *TCM operation in TCM_capture mode (lead frame complete)*

4.4.4 TCM_complete Mode

Once the capture frame is full, data logging stops and the TCM enters *TCM_complete* mode. The capture frame pointers are adjusted to span the entire TCM buffer.

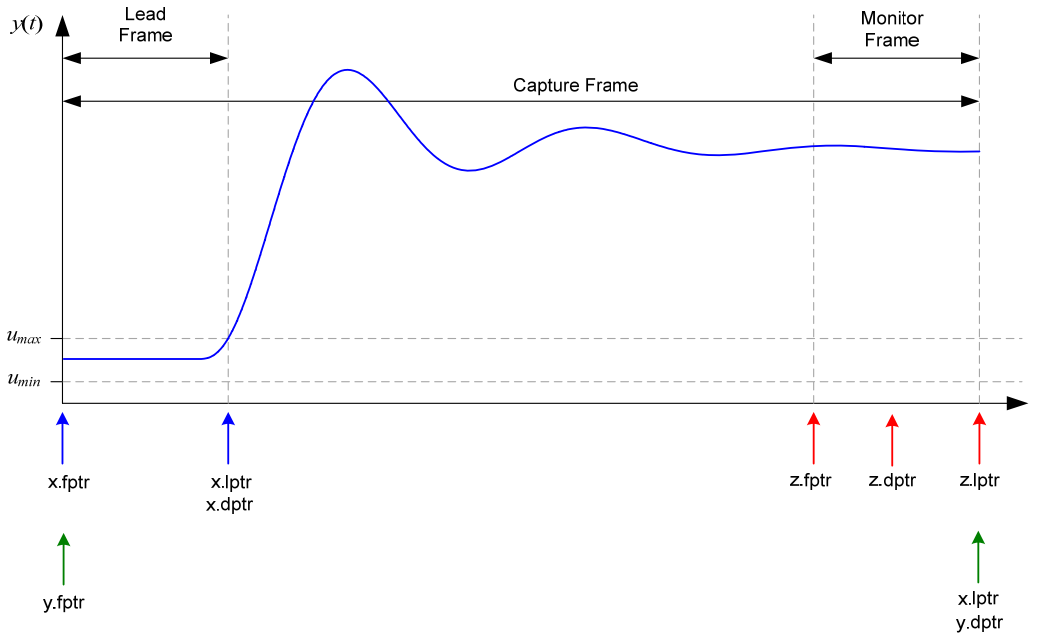


Figure 39. *TCM capture complete*

The buffer contents may now be read out using `DCL_readLog()` or `DCL_freadLog()`.

4.4.5 Functions

DCL_initTCM	***Initialize the TCM***

Header File: DCL_TCM.h

Source File: N/A

Declaration: void DCL_initTCM(TCM *q, float *addr, uint16_t size, uint16_t lead,
float tmin, float tmax)

Description: This function resets the TCM module. All buffer contents are loaded with zero, and the operating mode is set to “TCM_idle”.

Parameters:

q	The TCM structure
addr	The start address of the memory block
size	The size of the memory block in 32-bit words
lead	The length of the lead frame in samples
tmin	The upper trigger threshold
tmax	The lower trigger threshold

Return: Void

DCL_resetTCM	***Reset the TCM***

Header File: DCL_TCM.h

Source File: N/A

Declaration: void DCL_resetTCM(TCM *q)

Description: This function resets the TCM. The contents of the capture frame are loaded with zero. All data log pointers are re-initialized, and the operating mode is set to *TCM_idle*.

Parameters:

q	The TCM structure

Return: Void

DCL_armTCM	***Arm the TCM***

Header File: DCL_TCM.h

Source File: N/A

Declaration: uint16_t DCL_armTCM(TCM *q)

Description: If the current TCM mode is *TCM_idle*, this function changes it to *TCM_armed*, otherwise it is unchanged.

Parameters: q The TCM structure

Return: The current operating mode

DCL_runTCM	***Run the TCM***

Header File: DCL_TCM.h

Source File: N/A

Declaration: uint16_t DCL_runTCM(TCM *q, float data)

Description: Runs the TCM module.

Parameters: data The input data

q The TCM structure

Return: The current operating mode

4.5 Performance Measurement

4.5.1 Description

The Digital Control Library includes functions for the computation of control performance. All functions are based on discrete integration over a fixed interval of a variable representing servo error. The result is a non-negative scalar representing the quality of control: the smaller the result, the better the control. The figure below shows conceptually the transient servo error during a typical transient response.

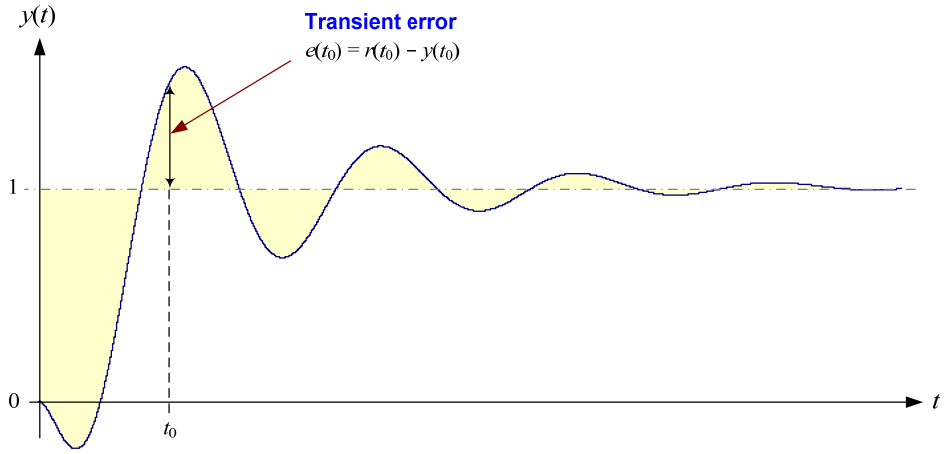


Figure 40. Transient servo error

There are three performance measures available in the library:

- The IES performance index is based on the square of the servo error. For an interval of N samples, with loop reference r and feedback y, the IES index (P_{IES}) is computed as follows.

Equation 48. $$P_{IES} = \sum_{k=1}^{N} (r(k) - y(k))^2$$

- The IAE performance index is based on the absolute value of the servo error. For an interval of N samples, with loop reference r and feedback y, the IAE index (P_{IAE}) is computed as follows.

Equation 49. $$P_{IAE} = \sum_{k=1}^{N} | r(k) - y(k) |$$

- The ITAE performance index is based on the time weighted absolute value of the servo error. For an interval of N samples, with loop reference r and feedback y, the ITAE index (P_{ITAE}) is computed as follows.

Equation 50. $$P_{ITAE} = \sum_{k=1}^{N} k | r(k) - y(k) |$$

Each index is available in two forms: one coded in assembly, the other in inline C. The computation time will depend on the length of the error log, but the assembly functions will always be significantly faster.

The following table shows cycle count benchmarks for each function, for a buffer of N data points. Cycle counts include function calling overhead from C.

Table 12. Performance index function benchmarks

Function	Cycles
IES_C1	24 + 6N
IES_C2	73 + 30N
IAE_C1	24 + 6N
IAE_C2	72 + 24N
ITAE_C1	26 + 7N
ITAE_C2	77 + 31N

4.5.2 Functions

DCL_runIES_C1	***Compute the IES Performance Index***

Header File: DCL_TCM.h

Source File: DCL_index.asm

Declaration: float DCL_runIES_C1(FDLOG *eLog)

Description: This function computes an IES performance index using the servo error data in a given memory block. The function is coded in assembly.

Parameters: eLog The servo error data log

Return: The IES index

DCL_runIES_C2 — ***Compute the IES Performance Index***

Header File: DCL_TCM.h

Source File: N/A

Declaration: float DCL_runIES_C2(FDLOG *eLog)

Description: This function is equivalent to DCL_runIES_C1, but is coded in inline C.

Parameters: eLog The servo error data log

Return: The IES index

DCL_runIAE_C1 — ***Compute the IAE Performance Index***

Header File: DCL_TCM.h

Source File: DCL_index.asm

Declaration: float DCL_runIAE_C1(FDLOG *eLog)

Description: This function computes an IES performance index using the servo error data in a given memory block. The function is coded in assembly.

Parameters: eLog The servo error data log

Return: The IAE index

DCL_runIAE_C2 — ***Compute the IAE Performance Index***

Header File: DCL_TCM.h

Source File: N/A

Declaration: float DCL_runIAE_C2(FDLOG *eLog)

Description: This function is equivalent to DCL_runIAE_C1, but is coded in inline C.

Parameters: eLog The servo error data log

Return: The IAE index

DCL_runITAE_C1 — ***Compute the ITAE Performance Index***

Header File: DCL_TCM.h

Source File: DCL_index.asm

Declaration: float DCL_runITAE_C1(FDLOG *eLog, float prd)

Description: This function computes an ITAE performance index using the servo error data in a given memory block. The function is coded in assembly.

Parameters: eLog The servo error data log

prd The sample period in seconds

Return: The ITAE index

DCL_runITAE_C2	***Compute the ITAE Performance Index***

Header File: DCL_TCM.h

Source File: N/A

Declaration: float DCL_runITAE_C2(FDLOG *eLog, float prd)

Description: This function is equivalent to DCL_runITAE_C1, but is coded in inline C.

Parameters: eLog The servo error data log

prd The sample period in seconds

Return: The ITAE index

4.6 Fixed Point Data Logger Support

4.6.1 Description

Version 2.1 of the Digital Control Library contains support for fixed point data. This comprises a set of data buffer functions, and a 4 channel data logger module. Both are similar to their floating point counterparts described earlier in this chapter. For this reason, neither will be described in detail here. The reader is referred to sections 4.2 and 4.3 for details.

4.6.2 Functions

DCL_deleteLog32	***Delete a Data Log***

Header File: DCL_log32.h

Source File: N/A

Declaration: void DCL_deleteLog32(LOG32 *p)

Description: This function resets all structure pointers to null value.

Parameters: p The LOG32 structure

Return: Void

DCL_resetLog32 — Reset a Data Log

Header File: DCL_log32.h

Source File: N/A

Declaration: void DCL_resetLog32(LOG32 *p)

Description: This function resets the data index pointer to start of the data log.

Parameters: p The LOG32 structure

Return: Void

DCL_initLog32 — Initialize a Data Log Structure

Header File: DCL_log32.h

Source File: N/A

Declaration: void DCL_initLog32(LOG32 *p, int32_t *addr, uint16_t size)

Description: This function assigns the buffer pointers to a memory block or array and sets the data index pointer to the first address.

Parameters: p The LOG32 structure

addr The start address of the memory block

size The length of the memory block in 32-bit words

Return: Void

DCL_writeLog32 — Write Data into a Log

Header File: DCL_log32.h

Source File: N/A

Declaration: int32_t DCL_writeLog32(LOG32 *p, int32_t data)

Description: This function writes a data point into the buffer and advances the indexing pointer, wrapping if necessary. The function returns the data value being over-written, which allows simple implementation of a fixed-length delay line.

Parameters: p The LOG32 structure

data The input data value address

Return: The over-written data value

DCL_fillLog32 — Fill a Data Log with Specified Data

Header File: DCL_log32.h

Source File: N/A

Declaration: void DCL_fillLog32(LOG32 *p, int32_t data)

Description: This function fills the data log with a given data value and resets the data index pointer to the start of the log.

Parameters:

p	The LOG32 structure
data	The fill data value

Return: Void

DCL_clearLog32 — Fill a Data Log Contents with Zero

Header File: DCL_log32.h

Source File: N/A

Declaration: void DCL_clearLog32(LOG32 *p)

Description: This function clears the buffer contents by writing 0 to all elements and resets the data index pointer to the start of the log.

Parameters:

p	The LOG32 structure

Return: Void

DCL_readLog32 — Fill a Data Log Contents with Zero

Header File: DCL_log32.h

Source File: N/A

Declaration: int32_t DCL_readLog32(LOG32 *p)

Description: This function reads a data point from the buffer and then advanced the index pointer, wrapping if necessary.

Parameters:

p	The LOG32 structure

Return: The indexed data value

DCL_copyLog32 — Copies one Data Log into Another

Header File: DCL_log32.h

Source File: N/A

Declaration: void DCL_copyLog32(LOG32 *p, LOG32 *q)

Description: This function copies the contents of one log into another and resets both buffer index pointers. The function assumes both logs have the same length.

Parameters:

p	The destination LOG32 structure
q	The source LOG32 structure

Return: Void

DCL_initMLOG32 — ***Initialize the MLOG32 Module***

Header File: DCL_MLOG32.h

Source File: N/A

Declaration: void DCL_initMLOG32(MLOG32 *q, int32_t *addr, uint16_t size, int32_t tmax, int32_t tmin, uint16_t div)

Description: This function loads all buffer contents with zero, and resets all buffer data pointers to their respective start addresses. The upper and lower trigger thresholds, and sample time scale are loaded into the MLOG32 structure. On completion, the MLOG32 module is in *MLOG32_idle* mode. This function is similar to `DCL_initMLOG()`.

Parameters:

q	The MLOG32 structure
addr	Pointer to the start address of the MLOG32 buffers
size	The size in 32-bit words of each buffer
tmax	The upper trigger threshold
tmin	The lower trigger threshold
div	The sample scaler

Return: Void

DCL_resetMLOG32 — ***Reset the MLOG32 Module***

Header File: DCL_MLOG32.h

Source File: N/A

Declaration: void DCL_resetMLOG32(MLOG32 *q)

Description: This function loads all buffer contents with zero, and resets all buffer data pointers to their respective start addresses. On completion, the MLOG32 module is in *MLOG32_idle* mode. This function is similar to `DCL_resetMLOG()`.

Parameters:

q	The MLOG32 structure

Return: Void

DCL_armMLOG32	***Arm the MLOG32 Module***

Header File: DCL_MLOG32.h

Source File: N/A

Declaration: void DCL_armMLOG32(MLOG32 *q)

Description: This function changes the MLOG32 operating mode from *MLOG32_idle* to *MLOG32_armed*. If the operating mode is not already *MLOG32_idle* when the function is called, it sets the mode to *MLOG32_idle*. This function is similar to `DCL_armMLOG()`.

Parameters: q The MLOG32 structure

Return: Void

DCL_runMLOG32	***Run the MLOG32 Module***

Header File: DCL_MLOG32.h

Source File: N/A

Declaration: void DCL_runMLOG32(MLOG32 *p)

Description: This function runs the MLOG32 module. If in the *MLOG32_armed* mode, the MLOG32 monitors its first input to determine whether either threshold has been exceeded. If so, the module enter *MLOG32_capture* mode and sample collection begins. When the buffers are full, the operating mode is set to *MLOG32_complete*. This function is similar to `DCL_runMLOG()`.

Parameters: q The MLOG32 structure

Return: Void

4.7 Simulation Models

4.7.1 The DCL Block-set

Version 3.0 of the Digital Controller Library is supplied with a small block-set of Simulink models. The block-set may be found in the Simulink file `DCL.slx` in the `\models` sub-directory of the DCL installation path. The following DCL controllers are represented in the block-set.

- PID_C1
- PID_C3
- PI_C1
- PI_C3
- PI_C5

- NLPID_C1
- DF11_C1
- DF13_C1
- DF22_C1
- DF23_C1

It is important to understand that the blocks capture the functional structure of the controller and are intended to be used for the purposes of control loop simulation only. None of the models have been configured for automated C code generation from Simulink or Matlab. Should the user wish to do this, it is their responsibility to re-configure and build the model accordingly. Code generation currently lies outside the scope of the DCL.

4.7.2 Simulation Example

An example Simulink model is included with the DCL. The example files are located in the `\models` sub-directory of the DCL installation path. To see the example, open and run the script file `DCL_example.m` in Matlab.

The script will open a configuration script file `PID_config.m`, which contains the control loop settings, including the PID controller gains. These will be manually adjusted by the user each time the example is run. After the configuration is loaded into the Matlab workspace, the script then loads the input stimulus from the file `PID_inputs.m`. The user can select the type of stimulus by changing the `input_config` variable near the top of the example script file.

The example opens the Simulink model `PID_sim.slx`. The example model consists of a simple feedback control loop using a PID_C1 controller from the DCL block-set. The plant is a third order transfer function with one LHP zero (see lines 22-30 in the configuration script file).

The simulation uses configuration parameters and input data in the Matlab workspace, and saves simulation data back to the workspace. The user may select which data to plot by setting plot variables in lines 20-25 of the example script. Plotting is performed in the script file `PID_plots.m`.

The example computes a performance index from the loop error and displays it in the Matlab command window. This is helpful should the user with to experiment with different controller parameters to improve performance.

Any data the user wishes to save is stored in data files in the `\models` sub-directory. Each data file contains a TI header line which facilitates loading the data onto a C2000 target device.

4.8 Double Precision Data Logging Functions

4.8.1 Description

Version 3.0 of the Digital Control Library adds a double precision floating-point data logger. Apart from data type, the data logger is similar to that described in section 4.2.

The data logger is supplied in the form of a C header file and may be used on any C2000 device irrespective of whether the DCL is used. The data logger is not compatible with the CLA.

The data logger operates with arrays of 64-bit floating-point data. The location, size, and indexing of each array are defined by three pointers capturing the start address, end address, and data index address. All three pointers are held in a common C structure with the data type “FDLOG64”, defined as follows:

```
typedef volatile struct {
      float64_t *fptr;
      float64_t *lptr;
      float64_t *dptr;
} FDLOG64;
```

Conceptually, the relationship between the array pointers and the elements of a data array of length “N” is shown below:

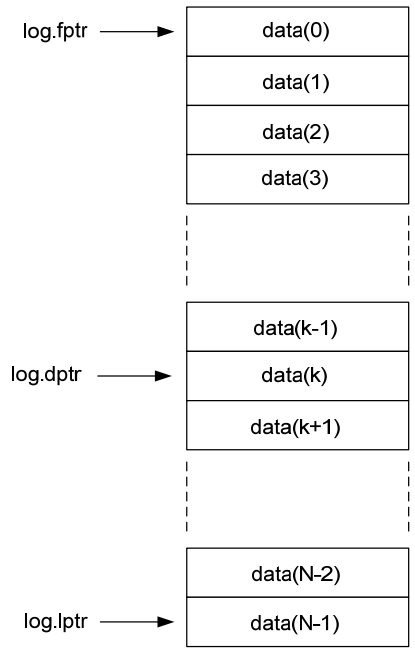


Figure 41. FDLOG64 pointer allocation

The data index pointer (dptr) always points to the next address to be written or read, and advances through the memory block as each new data value is written into the log. On reaching the end of the log, the pointer is reset to the first address in the log. The data logger header file contains a set of in-line C functions to access and manipulate data logs.

To use the data logger, you must include the header file `DCL_fdlog64.h` in your project. Typically, a user would create an instance of an FDLOG64 structure as follows:

```
FDLOG64 myBuf = FDLOG64_DEFAULTS;
```

The log pointers can then be initialized in the user’s code such that they reference a memory block in a specific address range. Thereafter, the code can clear or load the

buffer a specific data value, and then begin writing data into it using the `DCL_writeLog64()` function.

The execution cycles for the read & write functions are shown below:

Table 13. FDLOG64 read/write benchmarks

DCL_writeLog64	75
DCL_readLog64	58

4.8.2 Functions

DCL_deleteLog64	***Delete a Data Log***

Header File: DCL_fdlog64.h

Source File: N/A

Declaration: void DCL_deleteLog64(FDLOG64 *p)

Description: This function resets all structure pointers to null value.

Parameters: p The FDLOG64 structure

Return: Void

DCL_resetLog64	***Reset a Data Log***

Header File: DCL_fdlog64.h

Source File: N/A

Declaration: void DCL_resetLog64(FDLOG64 *p)

Description: This function resets the data index pointer to start of the data log.

Parameters: p The FDLOG64 structure

Return: Void

DCL_initLog64	***Initialize a Data Log Structure***

Header File: DCL_fdlog64.h

Source File: N/A

Declaration: void DCL_initLog64(FDLOG64 *p, float64_t *addr, uint16_t size)

Description: This function assigns the buffer pointers to a memory block or array and sets the data index pointer to the first address.

Parameters:

p	The FDLOG64 structure
addr	The start address of the memory block
size	The length of the memory block in 64-bit words

Return: Void

DCL_writeLog64 — ***Write Data into a Log***

Header File: DCL_fdlog64.h

Source File: N/A

Declaration: float64_t DCL_writeLog64(FDLOG64 *p, float64_t data)

Description: This function writes a data point into the buffer and advances the indexing pointer, wrapping if necessary. The function returns the data value being over-written, which allows simple implementation of a fixed-length delay line.

Parameters:

p	The FDLOG64 structure
data	The input data value address

Return: The over-written data value

DCL_fillLog64 — ***Fill a Data Log with Specified Data***

Header File: DCL_fdlog64.h

Source File: N/A

Declaration: void DCL_fillLog64(FDLOG64 *p, float64_t data)

Description: This function fills the data log with a given data value and resets the data index pointer to the start of the log.

Parameters:

p	The FDLOG64 structure
data	The fill data value

Return: Void

DCL_clearLog64 — ***Fill a Data Log Contents with Zero***

Header File: DCL_fdlog64.h

Source File: N/A

Declaration: void DCL_clearLog64(FDLOG64 *p)

Description: This function clears the buffer contents by writing 0 to all elements, and resets the data index pointer to the start of the log.

Parameters: p The FDLOG64 structure

Return: Void

DCL_readLog64 — ***Read Data from the Data Log***

Header File: DCL_fdlog64.h

Source File: N/A

Declaration: float64_t DCL_readLog64(FDLOG64 *p)

Description: This function reads one data point from the buffer and then advances the index pointer, wrapping to the first element if necessary.

Parameters: p The FDLOG64 structure

Return: The indexed data value

DCL_copyLog64 — ***Copies one Data Log into Another***

Header File: DCL_fdlog64.h

Source File: N/A

Declaration: void DCL_copyLog64(FDLOG64 *p, FDLOG64 *q)

Description: This function copies the contents of one log into another and resets both buffer index pointers. The function assumes both logs have the same length.

Parameters: p The destination FDLOG64 structure address

q The source FDLOG64 structure address

Return: Void

Chapter 5

Examples

This chapter describes the example projects supplied with the Digital Control Library.

Section

5.1 Example 1: DF22 compensator running on FPU32

5.2 Example 2: DF23 compensator running on CLA

5.3 Example 3: NLPID controller running on FPU32

5.4 Example 4: PI controller running on CLA

5.5 Example 5: PID controller running on FPU32

5.6 Example 6: TCM running on FPU32

5.7 Example 7: Smith predictor running on FPU32

5.8 Example 8: GSM running on FPU32

5.9 Example 9: Multiple Controller System with ERAD running on FPU32

The Digital Control Library package includes a set of code examples intended to illustrate the use of library functions in a typical software project. Examples are supplied as CCS projects configured for use with either the F28069 or the F280049 device, and will run on any appropriate target board. Code migration to a different C2000 device is straightforward and does not affect the DCL.

There are nine example projects, located in the C2000Ware installation directory, in the sub-directory `\libraries\control\DCL\c28\examples`. Each example has a "CCS" sub-directory containing a ".projectspec" file which should be imported as a project into the users CCS workspace.

The following sections describe the example code and outline the steps to run them. The examples were prepared using CCS version 6. It is assumed the reader is familiar with CCS and how to build and run code. For further information on these topics, the reader is referred to the C2000 training workshop (section 6.2).

5.1 Example 1: DF22 Compensator Running on FPU32

5.1.1 Example Overview

This example demonstrates the DF22 compensator running on the FPU32 core. The code creates two separate instances of the DF22 compensator: one implemented using the full DCL_DF22_C1 function, the other using the pre-computed DCL_DF22_C2 and DCL_DF22_C3 functions. The pre-computed compensator makes use of a clamp function to limit the compensator output.

The program contains one ISR which is triggered by a CPU timer at 1kHz. The ISR reads a single input from a data buffer and runs both DF22 compensators. The compensator outputs are compared, and then both outputs and their difference logged into three separate data buffers. When the last point of the input buffer has been read and processed, the ISR passes through the line containing a `NOP` instruction near the bottom of the program. The user can place a break-point here to examine the results of the compensator test.

The program makes use of four data buffers at the following addresses:

- 0xC000 – contains input data representing servo loop error
- 0xE000 – contains output data from the full DF22 compensator
- 0x10000 – contains output data from the pre-computed DF22 compensator
- 0x12000 – contains the difference between the two compensator outputs

Each data buffer contains 1601 single precision floating-point data points.

5.1.2 Code Description

The following lines in the program files `Example_F28069_DF22.c` are important:

Lines 9-29: create four data buffers and assign them to memory blocks defined in the linker file `F28069_DCL.cmd`

Lines 43-44: create instances of the two DF22 compensators

Lines 66-72: initialize the data log structures and data buffers

Lines 75-86: initialize the coefficients of the two compensators

Lines 89-90: set the clamp limits for the pre-computed compensator

Line 115: tests whether the last element in the input data buffer has been reached

Line 118: reads the input data point

Line 121: runs the full DF22 compensator

Line 124: runs the immediate part of the pre-computed compensator

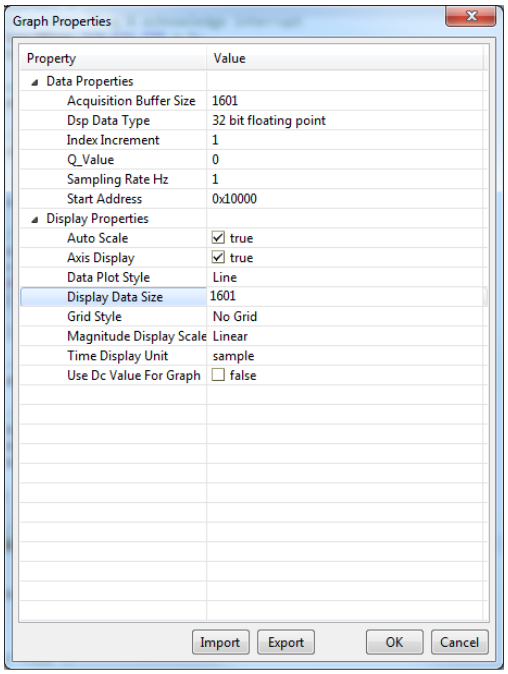
Line 125 clamps the output of the pre-computed compensator

Lines 126-129: run the partial part of the pre-computed compensator

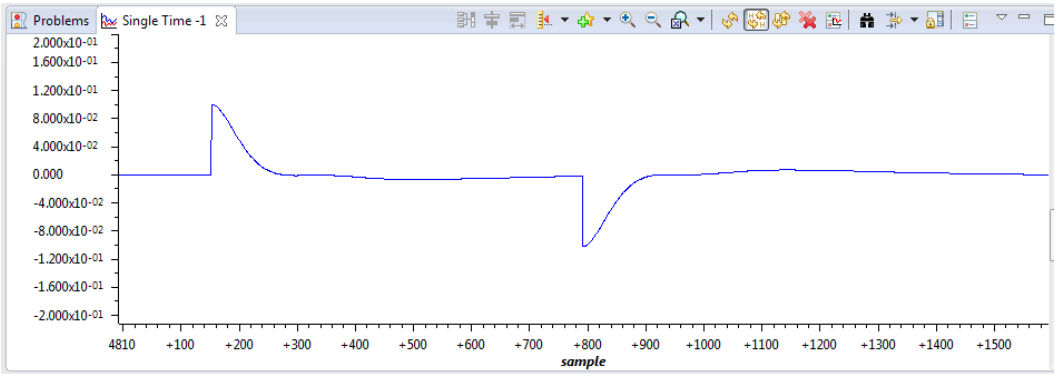
5.1.3 Running the Example

To run this example, first build and load the program onto the C28x device, then load the data file `DF22_edata.dat` into data memory at address 0xC000. This file contains a pre-recorded data sequence representing simulated servo loop error at the controller input.

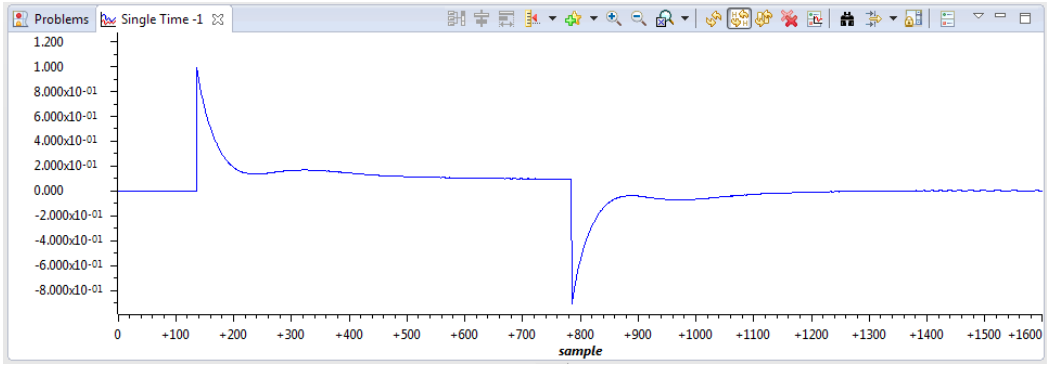
Place a break-point at the line indicated in the control ISR and run the program. When the program reaches the break-point, inspect the memory buffers by opening a CCS graph window. The graph setup window below shows how to configure the graph to view the pre-computed compensator output (u2k).



The ek buffer should look like this.



The plot of the `u1k` and `u2k` buffers should look like this.



Similar graphs can be opened for the other buffers by changing the start address. The buffer at start address 0x12000 captures the difference in output between the full and pre-computed compensators, and should contain data which is zero or very small.

5.2 Example 2: DF23 Compensator Running on CLA

5.2.1 Example Overview

This example demonstrates one method of running a DF23 compensator on the CLA. In this example, incoming data is read in an ISR on the C28x CPU and passed to the CLA using a variable (ek) which is located in CPU-to-CLA message RAM. The ISR then triggers task 3 on the CLA and waits for it to complete.

The CLA task calls two different DF23 compensators and stores their results in two variables (u1k and u2k) which are located in CLA-to-CPU message RAM. These results are read by the CPU ISR which computes the difference between them. The ISR then stores both results and their difference to three data buffers. When the final input value has been processed in this way the ISR passes through a `NOP` instruction, allowing the user to place a break-point and inspect the results.

5.2.2 Code Description

The following lines in the file `Example_F28069_DF23.c` are important:

Lines 11-31: create four buffers which will be used to hold control data

Lines 37-42: create variables which will pass control data between C28x and CLA

Lines 68-74: assign data logs to the buffers and initialize them

Line 141: reads the next data point from the input data file

Line 145: triggers the CLA task which will call the DF23 compensator

Line 149: compute the difference between the two compensator results

Lines 152-154: write the compensator results into the data buffers

The following lines in the file `F28069_DF23_CLA.cla` are important:

Line 31: calls the full DF23 compensator and stores the result in u1k

Line 34: calls the immediate DF23 compensator and stores the result in u2k

Line 35: clamps the immediate result and sets the clamp flag vk

Lines 36-39: pre-compute the next partial DF23 result, providing the immediate part is in range

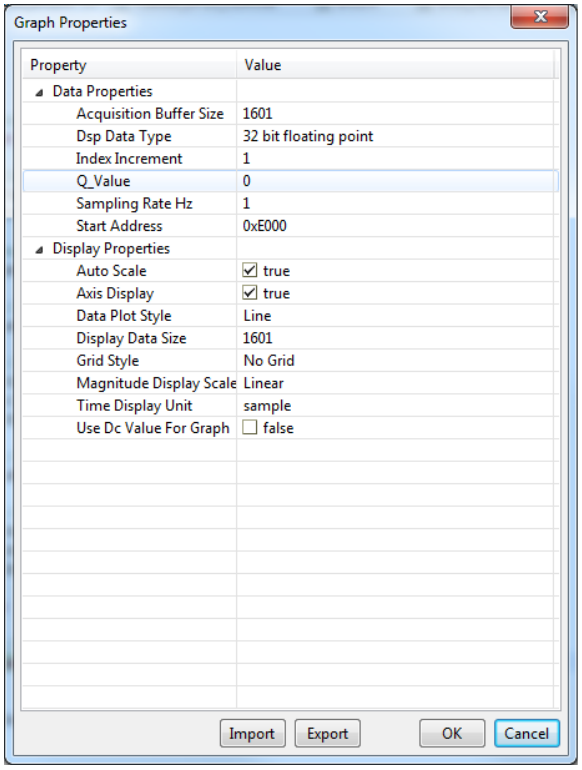
Lines 72-92: initialize the two DF23 compensator structures

Lines 95-96: initialize the clamp limits for the pre-computed DF23 compensator

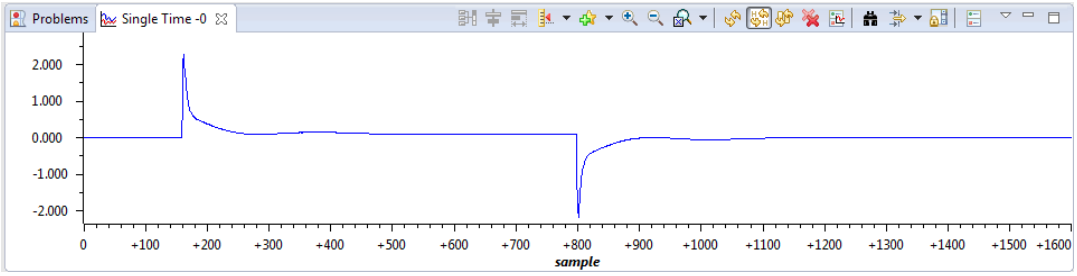
5.2.3 Running the Example

To run this example, first build the project, then load the project onto the C28x and load symbols onto the CLA. Load the pre-recorded data file `DF23_edata.dat` into C28x memory at address 0xC000. Place a break-point at the `NOP` instruction in line 159 of the file `Example_F28069_DF23.c`, and run the program.

After the break-point is reached, open a graph window to inspect the contents of the u1k memory buffer at address 0xE000.



The graph contents should look like this.



These results were produced by the full DF23 compensator. To view the pre-computed compensator, change the "Start Address" field in Graph Properties to 0x10000. The buffer at address 0x12000 contains the difference in compensator results, all of which should be zero or very small.

5.3 Example 3: NLPID Controller Running on FPU32

5.3.1 Example Overview

This example demonstrates the use of the non-linear PID controller on the FPU32 core. The code is similar to the linear PID example below except that use is made of the SPS and CSS sub-structures to update controller parameters in the background loop. Note that the program includes `DCL_NLPID.h` which contains the controller functions.

The ADC is triggered by a PWM zero match event, and samples two channels. An ISR is triggered by an ADC end-of-conversion event and reads the ADC results. Channel A0 represents the control loop feedback, and channel B0 represents an external saturation input which is used for integrator anti-windup. The `DCL_runClamp_C2` function is used to convert the ADC result into an integer with the logical value 1 or zero.

The ISR calls the DCL_NLPID_C1 controller, and stores the control effort in the uk variable. This is converted into an unsigned 16-bit integer and used to modulate the PWM duty cycle. In this way, the program implements a non-linear closed loop controller which regulates the floating-point reference, rk.

5.3.2 Code Description

The following lines in the file `Example_F28069_NLPID.c` are important:

Lines 25-27: creates instance of the NLPID controller ant its' sub-structures

Lines 103-104: assign the substructure addresses to the active controller structure

Lines 106-123: initialize the controller parameters in SPS shadow structure

Lines 126-127: copy the SPS parameters into the active controller structure

Line 146: updates active controller parameters if an update request is pending

Line 162: reads the control feedback and converts to signed floating-point format

Line 163: converts the saturation input to 0.0f or 1.0f for anti-windup reset

Line 169: calls the non-linear PID controller function

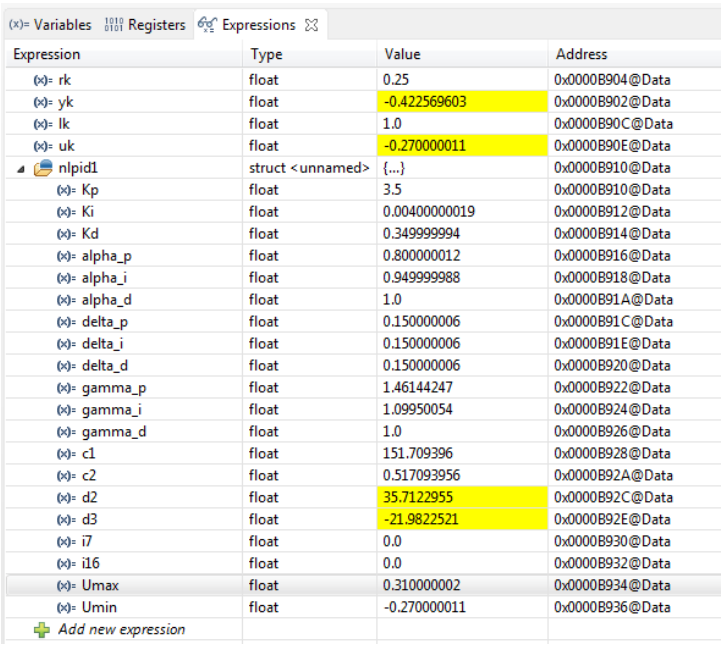
Line 172: converts the controller output to unsigned 16-bit integer

Line 173: updates the PWM duty cycle

5.3.3 Running the Example

To run this example, simply build, load, and run the program on the C28x core. Place a break-point at the last instruction in the ISR (line 164) and run the program. The following variables can be monitored in a watch window:

- rk – input reference
- yk – feedback
- lk – external saturation flag
- uk – controller output
- nlpid1 – the NLPID controller structure



Expression	Type	Value	Address
rk	float	0.25	0x0000B904@Data
yk	float	-0.422569603	0x0000B902@Data
lk	float	1.0	0x0000B90C@Data
uk	float	-0.270000011	0x0000B90E@Data
nlpid1	struct <unnamed>	{...}	0x0000B910@Data
Kp	float	3.5	0x0000B910@Data
Ki	float	0.00400000019	0x0000B912@Data
Kd	float	0.349999994	0x0000B914@Data
alpha_p	float	0.800000012	0x0000B916@Data
alpha_i	float	0.949999988	0x0000B918@Data
alpha_d	float	1.0	0x0000B91A@Data
delta_p	float	0.150000006	0x0000B91C@Data
delta_i	float	0.150000006	0x0000B91E@Data
delta_d	float	0.150000006	0x0000B920@Data
gamma_p	float	1.46144247	0x0000B922@Data
gamma_i	float	1.09950054	0x0000B924@Data
gamma_d	float	1.0	0x0000B926@Data
c1	float	151.709396	0x0000B928@Data
c2	float	0.517093956	0x0000B92A@Data
d2	float	35.7122955	0x0000B92C@Data
d3	float	-21.9822521	0x0000B92E@Data
i7	float	0.0	0x0000B930@Data
i16	float	0.0	0x0000B932@Data
Umax	float	0.310000002	0x0000B934@Data
Umin	float	-0.270000011	0x0000B936@Data
Add new expression			

The user can run repeatedly to the break-point, modifying controller parameters and examining the change of controller variables. If any of the “alpha” or “delta” parameters are changed, the variable calFlag should be set to 1 to enable the “gamma” gains to be computed and updated in the background loop.

5.4 Example 4: PI Controller Running on CLA

5.4.1 Example Overview

This example demonstrates one method of running a PI controller on the CLA. The CPU program contains an ISR which is triggered by an ADC end-of-conversion in the same way as example 3. Feedback data is read from the ADC in an ISR on the C28x CPU and passed to the CLA using a variable (yk) which is located in CPU-to-CLA message RAM, together with the servo reference (rk). The ISR converts the ADC result into signed floating-point format, then triggers task 3 on the CLA and waits for it to complete.

CLA task 3 calls the function `DCL_runPI_L1()` which computes the PI controller in an assembly function. The result is stored in the variable uk, which is located in CLA-to-CPU message RAM.

The PI controller result is read by the ISR, converted into a scaled un-signed 16-bit integer, and written to the PWM duty cycle register.

5.4.2 Code Description

The following lines in the file `Example_F28069_PI.c` are important:

Lines 19-24: create instances of the control variables and assign them to the appropriate message RAM blocks

Lines 26-27: create an instance of the PI controller structure and place it in CPU-to-CLA message RAM. This allows controller parameters to be modified from code running on the C28x CPU.

Lines 44-49: initialize the PI controller parameters

Line 168: reads the feedback data from the ADC and converts it into floating-point format

Line 172: starts CLA task 3 and waits for it to complete

Lines 175-176: convert the controller result to 16-bit unsigned integer and write it to the PWM duty cycle register

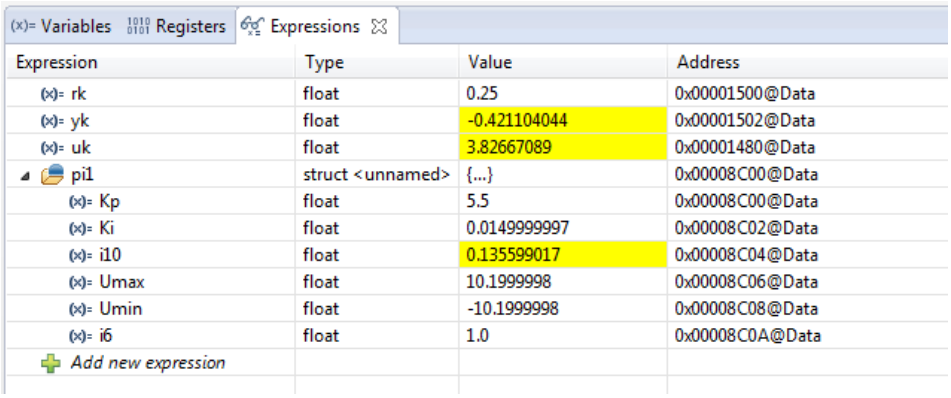
The following line in the file `F2806x_PI_CLA.cla` is important:

Line 24: calls the PI controller function `DCL_runPI_L1()`

Note that in this example, initialization of the PI controller is performed on the C28x CPU, so there is no need to allocate a separate CLA task for that purpose.

5.4.3 Running the Example

Build and load the project onto the C28x, then load the symbols onto the CLA. Place a break-point at the last instruction in the ISR (line 178), and run the program. Open an Expressions Window in CCS, and inspect the control variables and PI controller structure.



Expression	Type	Value	Address
rk	float	0.25	0x00001500@Data
yk	float	-0.421104044	0x00001502@Data
uk	float	3.82667089	0x00001480@Data
pi1	struct <unnamed>	{...}	0x00008C00@Data
Kp	float	5.5	0x00008C00@Data
Ki	float	0.0149999997	0x00008C02@Data
i10	float	0.135599017	0x00008C04@Data
Umax	float	10.1999998	0x00008C06@Data
Umin	float	-10.1999998	0x00008C08@Data
i6	float	1.0	0x00008C0A@Data
Add new expression			

At this point, controller gains can be manually changed and the code run repeatedly to observe the effect on the control variables.

5.5 Example 5: PID Controller Running on FPU32

5.5.1 Example Overview

This simple example demonstrates a common digital control scenario: a single linear PID controller running on the FPU32 core which reads an ADC channel and manipulates PWM duty cycle.

The example project contains an ISR which is triggered by an ADC end-of-conversion event. The ADC is triggered by a PWM zero match event, and samples two channels which are read by the ISR. Channel A0 represents the control loop feedback, and channel B0 represents an external saturation input which is used for integrator anti-windup. The `DCL_runClamp_C1` function is used to convert the ADC result into an integer with the logical value 1 or zero.

The ISR calls the DCL_PID_C4 parallel form PID controller to compute control effort held in the “uk” variable. This is converted into an unsigned 16-bit integer and used to modulate the PWM duty cycle. In this way, the program implements a simple closed loop PWM controller which regulates the floating-point reference, “rk”.

5.5.2 Code Description

The following lines in the example file `Example_F28069_PID.c` are important:

Lines 94-105: initializes the elements of the PID structure

Line 107: sets the reference input to the control loop

Line 108: initializes the external saturation flag

Line 137: reads the feedback and converts to the range ±1.0f

Line 138: reads the external saturation variable lk

Line 141: converts lk to 1.0f or 0.0f

Line 144: runs the PID controller

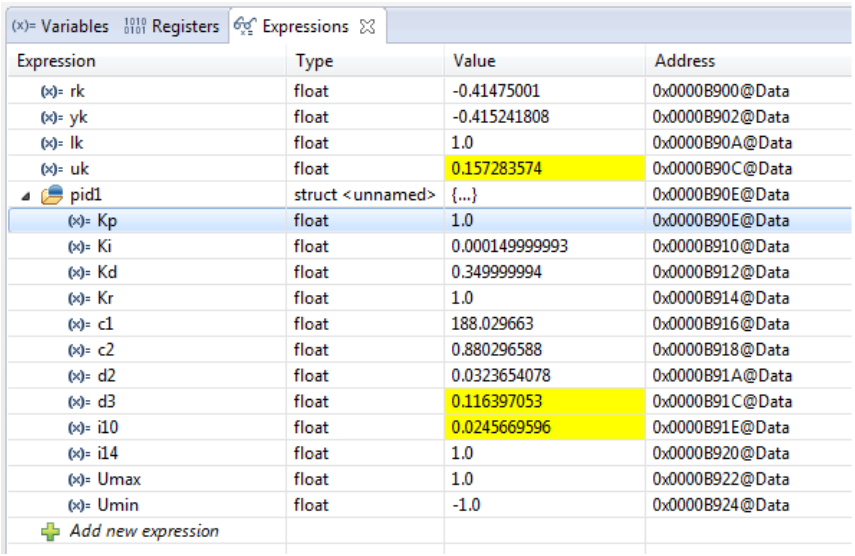
Line 147: convert the controller output to unsigned integer in the range 0 to PRD

Line 148: write result to PWM duty cycle register

5.5.3 Running the Example

To run this example, simply build, load, and run the program on the C28x core. Place a break-point at the last instruction in the ISR and run the program (line 150). The following variables can be monitored in a watch window:

- rk – input reference
- yk – feedback
- lk – external saturation flag
- uk – controller output
- pid1 – the PID controller structure



Expression	Type	Value	Address
(x)= rk	float	-0.41475001	0x0000B900@Data
(x)= yk	float	-0.415241808	0x0000B902@Data
(x)= lk	float	1.0	0x0000B90A@Data
(x)= uk	float	0.157283574	0x0000B90C@Data
pid1	struct <unnamed>	{...}	0x0000B90E@Data
(x)= Kp	float	1.0	0x0000B90E@Data
(x)= Ki	float	0.000149999993	0x0000B910@Data
(x)= Kd	float	0.349999994	0x0000B912@Data
(x)= Kr	float	1.0	0x0000B914@Data
(x)= c1	float	188.029663	0x0000B916@Data
(x)= c2	float	0.880296588	0x0000B918@Data
(x)= d2	float	0.0323654078	0x0000B91A@Data
(x)= d3	float	0.116397053	0x0000B91C@Data
(x)= i10	float	0.0245669596	0x0000B91E@Data
(x)= i14	float	1.0	0x0000B920@Data
(x)= Umax	float	1.0	0x0000B922@Data
(x)= Umin	float	-1.0	0x0000B924@Data
Add new expression			

The user can run repeatedly to the break-point modifying controller parameters and examining the change of controller variables.

5.6 Example 6: TCM Running on FPU32

5.6.1 Example Overview

This example illustrates the use of the TCM to capture a portion of a pre-recorded sample transient response. The code demonstrates how to configure and use the TCM, together with computation and storage of servo error, use of the fast read & write data log functions, and use of the performance index functions.

The code contains a single ISR triggered at 1kHz by a CPU timer. The ISR uses the fast read function `DCL_fread()` to read values from two buffers representing servo reference (rBuf), and feedback (yBuf), and then runs the TCM on the feedback sample to detect and capture the transient response in a third buffer (dBuf). The code subtracts yk from rk to find the instantaneous servo error, and logs the result into a fourth buffer (eBuf). When the final point in the input data sequence has been read and acted upon, the dBuf buffer should contain a portion of the feedback sequence around the transient edge, while the eBuf buffer contains the servo error. The variables P1, P2, and P3 contain the ITAE, IAE, and IES performance indices respectively. These are computed over the entire input sequence.

5.6.2 Code Description

The following lines in the file `Example_F28069_TCM.c` are important:

Lines 14-32: create instances of four data buffers and assign them to specific regions defined in the linker command file `F28069_DCL.cmd`.

Lines 39-41: create instances of the control variables

Line 42: creates an instance of the TCM module and initializes it to default values

Lines 43-45: creates variables to hold the performance indices

Lines 67-71: assign the data buffers to FDLOG structures and clear the servo error buffer

Lines 101-102: read the servo reference and feedback data

Line 105: runs the TCM

Lines 108-109: compute the servo error and log is to the eBuf data buffer

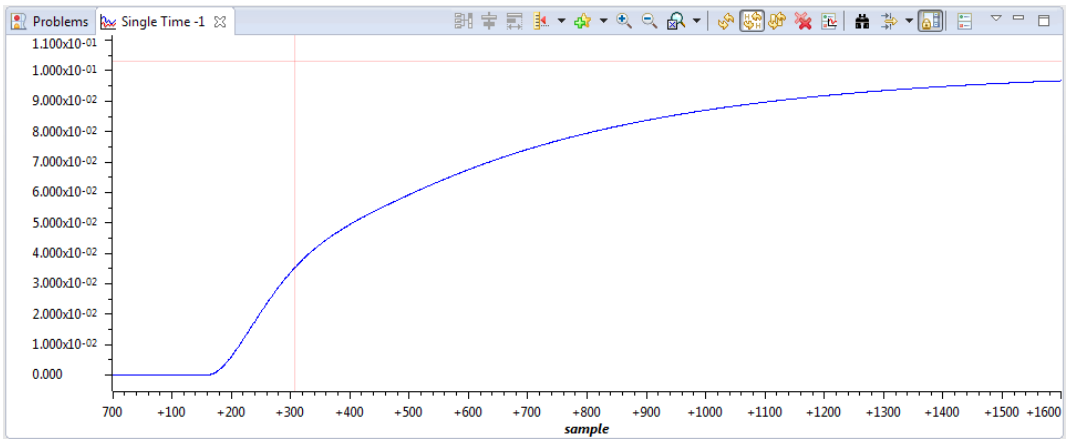
Line 112: detects if the final point in the input buffer has been processed

Lines 118-120: compute the three performance indices

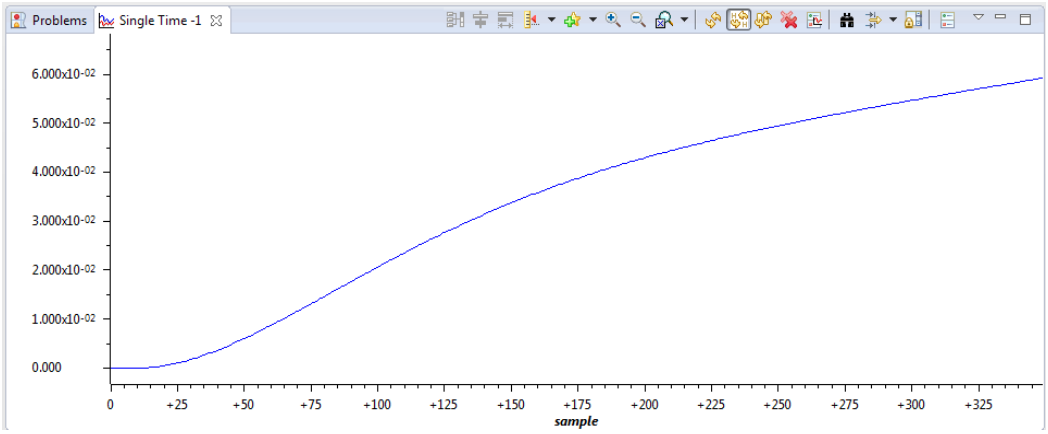
5.6.3 Running the Example

To run this program, build and load the project onto the target device. Then, load the two supplied data files `TCM_input.dat` and “TCM_response.dat” into data memory addresses 0xc000 and 0xe000 respectively. Place a break-point on the `NOP` instruction at the bottom of the ISR, and run the program.

When the break-point is reached, open a graph window to view the contents of the 1601-point yBuf memory at address 0xe000.



Open a second graph to display the 350-point contents of the dBuf buffer at address 0x12000.



The TCM buffer contains part of the feedback data near the transient. Notice that the first part of the TCM buffer (approximately 25 samples) contains data which has not exceeded either trigger threshold. This is the lead frame.

5.7 Example 7: Smith Predictor Running on FPU32

5.7.1 Example Overview

This example illustrates the simultaneous use of two direct form compensators, together with a time delay implemented using a data logger to construct a Smith predictor. The Smith predictor facilitates control of systems which incorporate a fixed time delay. To construct the controller, the plant and the time delay must be known since the controller includes models of both. The architecture of the basic Smith predictor is shown below.

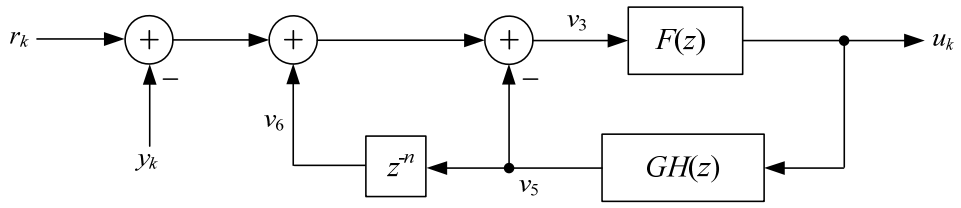


Figure 42. Smith Predictor control loop

In this example, the plant has a third order characteristic and the delay line is 13 sample periods in duration. The plant transfer function GH(z) is shown in the source code. The controller F(z) is also third order, implemented using a DF23 structure.

5.7.2 Code Description

The following lines in the file `Example_F28069_Smith.c` are important:

Lines 25-28: create an instance of the DF23 controller and sub-structures (note that the sub-structures are not used in this example).

Lines 34-37: create the delay line which we will use to model time delay in the plant

Lines 112-113: initialize the delay line and buffer pointers

Lines 121-128: initialize the plant model

Lines 131-137: initialize the controller

Lines 169-172: execute the controller

5.7.3 Running the Example

To run this program, build and load the project onto the target device. Then, set a break-point on line 169 and run the code. At this point the user may step through the example code and verify the controller and delay line are working as expected. For example, the variable v3 could be added to the CCS Expressions window and monitored as the program is run between break-points.

To verify the working of the delay model, add the variable v6 and the delay line array “p_array” to the expressions window, then set a break-point at the end of the ISR and observe the data updates in the array.

5.8 Example 8: GSM Running on FPU32

5.8.1 Example Overview

This example illustrates the setup and use of the Gain Scheduler Module running on an F28069. The code makes no use of the device peripherals.

The example is configured to construct a $y = x^2$ target profile. Offsets and gains are loaded into the SPS, and the update performed using the `DCL_updateGSM()` function.

5.8.2 Code Description

The following lines in the file `Example_F28069_GSM.c` are important:

Lines 23-25: create instances of the GSM module and supporting sub-structures, initialized with their default values.

Lines 41-42: Initialize the results buffer which will hold the input-output curve.

Lines 45-46: load the GSM sub-structure addresses.

Lines 49-54: loads the GSM shadow offset table from a target function.

Line 57: loads the shadow gain table from the shadow offsets.

Lines 60-61: copy the shadow parameter sets into the active controller tables.

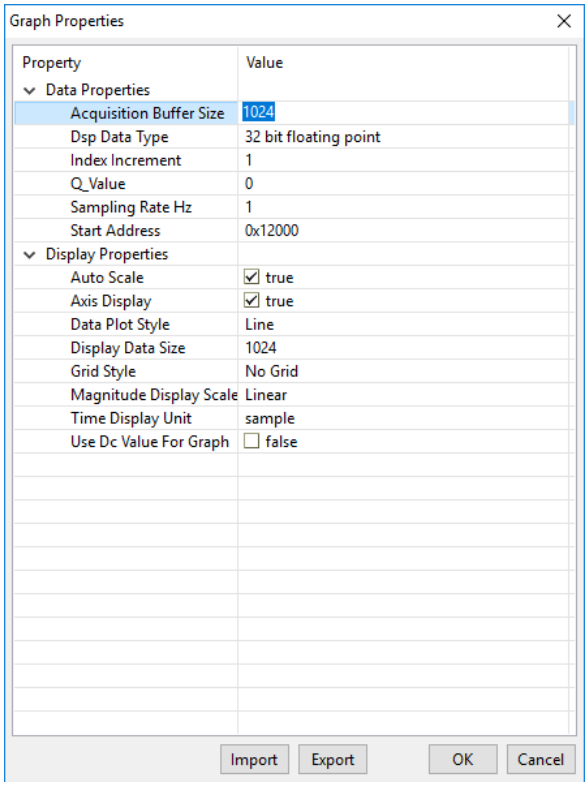
Lines 64-69: runs the GSM to construct the input-output table and stores it in the results buffer.

Lines 80-84: contain the target profile function.

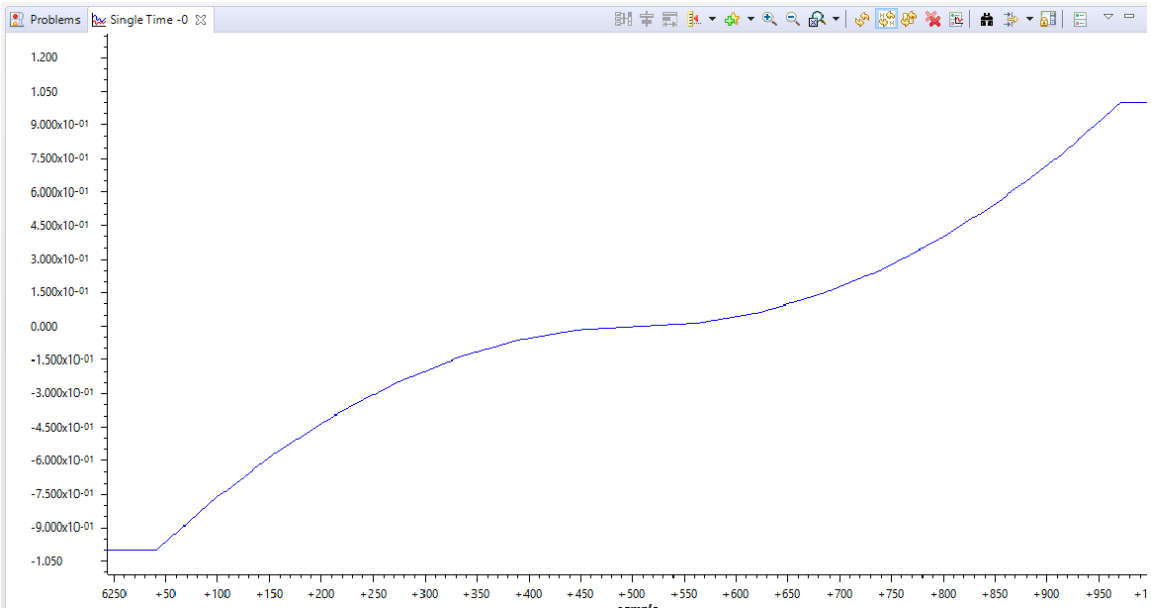
5.8.3 Running the Example

To run this program, build and load the project onto the target device. Then, set a break-point at the NOP instruction in the while loop at line 74.

Open a graph window to view the contents of the result buffer. The graph properties window should be configured as follows.



The graph shows the GSM input-output curve and for the default target function ($y = x^2$) should look like this:



The user is invited to experiment with different target functions.

5.9 Example 9: Multiple Controller System with ERAD Running on FPU32

5.9.1 Example Overview

This example illustrates the simultaneous use of multiple DCL controllers running at different update rates on an F280049. This device contains debug hardware known as Embedded Real-time Analysis and Diagnostics (ERAD) which is capable of profiling the system without having to connect an emulator. Among other things, the ERAD can measure the execution time of a specified code function. The example demonstrates use of ERAD to determine CPU bandwidth in a complex control application consisting of three separate controller interrupts, as follows.

ISR	Rate (kHz)	CPU Timer
a	10	1
b	253	2
c	60	3

ISR (a) reflects a situation common in field oriented motor control in which a relatively low rate PID controller, possibly regulating shaft speed, is cascaded with two parallel PI current control loops running approximately ten times faster. The interrupt runs at 10 kHz, and the PID controller is “time sliced” to run ten times slower, at 1 kHz.

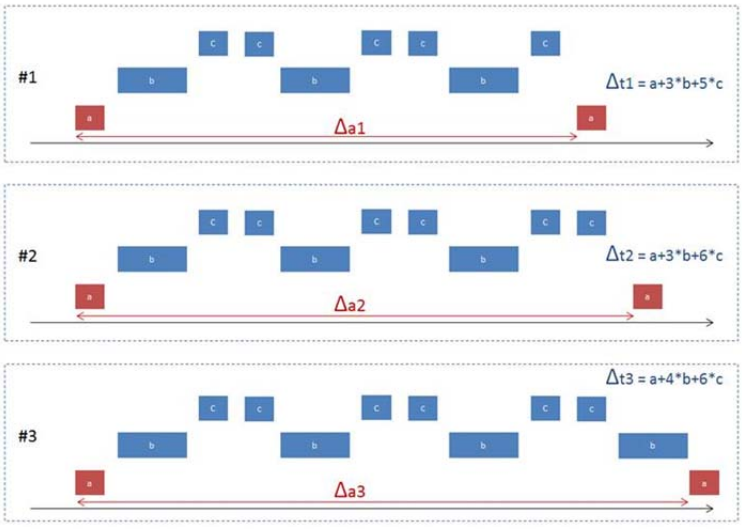
ISR (b) represents a higher rate loop, perhaps regulating other power electronics sub-systems such as PFC control. The ISR contains a gain scheduler and DF11 compensator in cascade, running at the full 253 kHz rate. Also in this ISR is a DF22 compensator which is “time sliced” to run at one third of the interrupt rate.

The last loop, ISR (c), contains a DF23 compensator and is configured with a varying payload, so that the execution time changes as the program runs. Also, the DF23 compensator coefficients are updated periodically in the background loop.

The overall program is fairly demanding from the point of view of computational load and the programmer might reasonably wish to know the CPU bandwidth to ensure hard real-time deadlines are being met. This information can be found using the hardware ERAD module. Refer to the ERAD chapter in the TMS320F28004x Technical Reference Manual for information on the ERAD module and registers.

This example uses the ERAD module to measure the CPU “bandwidth”: the amount of cycles available after all the application real-time requirements have been met. To determine this, we measure the number of CPU cycles between consecutive calls to the slowest interrupt in the system, and subtract from is the total number of cycles spent servicing interrupts over the same interval. This is equivalent to the number of cycles spent in the background loop over the longest interrupt interval and is usually expressed as a percentage.

In this example, we could have 3 or 4 ISR (b) calls and 5 or 6 ISR (c) calls between two consecutive ISR (a) calls.



With reference to the above figure, the example code measures the worst case CPU availability as follows:

```
MaxWindowCycles = Max(Δa1, Δa2, Δa3);
MaxISRCycles    = Max(Δt1, Δt2, Δt3);
```

Then we compute available CPU bandwidth in % using

CPU bandwidth = 100 x (`MaxWindowCycles` – `MaxISRCycles`) / `MaxWindowCycles`

5.9.2 Code Description

The example code makes use of an API function set to configure the ERAD module.

The example links all the ISR code to a single named section (“interruptSection”) for ease of profiling. Refer to the files `F280049_ERAD.cmd` and `F280049_stdmem.cmd` to see how this is done. Note that the method of configuring the ERAD module involves the use of API functions contained in the source file `f28004x_dcl_erad.c`.

The following lines in the file `Example_F280049_ERAD.c` are important.

Lines 8 – 39: create instances of the controllers and control variables used in the program, and define the constants used for decimation.

Lines 79 – 148: initialize the controller parameters.

Lines 168 – 169: update the DF23 controller parameters used in ISR (c).

Lines 174 – 187: tests the number of passes through the background loop and halts execution when the number exceeds (arbitrarily) 1000, after which CPU timers are stopped and the PC is trapped in an infinite “while” loop.

Lines 192 – 244: contain the three interrupt service routines, each of which is associated with the named section “interruptSection” using a CODE_SECTION pragma.

Lines 247 – 315: configure the ERAD registers using the API set to capture the information described in the previous section.

5.9.3 Running the Example

To run this program, build and load the project onto the target device. Then, place a break-point on line 186 of the file `Example_F280049_ERAD.c`.

Open an Expressions window in CCS and add the variables shown below, then run the program. The values should be similar to those shown below.

(x)= Variables | Expressions | Registers | Breakpoints

Expression	Type	Value
(x)= IsraCount	unsigned int	121
(x)= IsrbCount	unsigned int	1772
(x)= IsrcCount	unsigned int	469
(x)= IdleLoopCount	unsigned long	1000
(x)= MaxISRCycles	unsigned long	12136
(x)= MaxWindowCycles	unsigned long	19240

The best case CPU bandwidth measured over 1000 cycles is:

100 x (19240 – 12136) / 19240 = 36.92%

The user may like to experiment with the configuration of the ERAD module to make other measurements.

Chapter 6

Support

This chapter contains a list of useful technical resources relevant to the DCL.

Section

6.1 References

6.2 Training

6.3 Support

6.1 References

6.1.1 C2000 Documentation

Documentation for C2000 MCU devices can be found on their respective product pages at www.ti.com/c2000. The following references are relevant to the DCL.

- *TMS320C28x Assembly Language Tools* User's Guide
  http://www.ti.com/lit/ug/spru513m/spru513m.pdf
- *TMS320C28x Optimizing C/C++ Compiler* User's Guide
  http://www.ti.com/lit/ug/spru514m/spru514m.pdf
- *TMS320C28x CPU & Instruction Set* User's Guide
  http://www.ti.com/lit/ug/spru430f/spru430f.pdf
- *TMS320C28x Floating Point Unit & Instruction Set* Reference Guide
  http://www.ti.com/lit/ug/sprueo2b/sprueo2b.pdf
- *TMS320x2803x Piccolo Control Law Accelerator (CLA)* Reference Guide
  http://www.ti.com/lit/ug/spruge6b/spruge6b.pdf

6.1.2 Literature

The following is a selection of publications on control theory.

- *Feedback & Control Systems*

  J.J.DiStefano, A.R.Stubberud & I.J.Williams, Schaum, 2011
- *Digital Control of Dynamic Systems*

  G.F.Franklin, J.D.Powell & M.L.Workman, Addison-Wesley, 1998

- *Control Theory Fundamentals*

  R.Poley, CreateSpace, 2015

6.2 Training

- Training materials for the C2000 devices from Texas Instruments can be found at processors.wiki.ti.com/index.php/Category:C2000_Training
- Recordings of a 1-day hands-on workshop using the F28069 device can be found at training.ti.com/c2000-mcu-1-day-workshop-8-part-series
- Information on a series of technical seminars in control theory, including video recordings, can be found at https://sites.google.com/site/controltheoryseminars/

6.3 Support

Technical support on the DCL is available via the C2000 e2e forum, at e2e.ti.com/support/microcontrollers/c2000

Reports of errors or omissions, or suggestions for additions and improvements to the library, are always welcome via the e2e forum.

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265