Institut für Datentechnik
und Kommunikationsnetze

Technische Universität
Braunschweig

OSIRIS

Command Token Interpreter and UDP Manager

Revision 1.0

IDA-OCL-0003

January 2003

Prepared by
Tim Wittrock

Tables and Figures

1 Scope

1.1 Purpose of this Document

This document is a short description of the OSIRIS UDP token interpreter and the spacecraft part of the UDP manager. The interpreter is used to execute user-defined functions that preprocess data and control and operate the experiment environment. The UDP manager provides the link to ground and manages the UDPs.

1.2 Change Record

Table 1 Change Record

Date	Revision	Author	Affected Sections
02/28/2001	0.1	Wittrock	All sections
07/10/2001	0.2	Wittrock	UDP manager
07/25/2001	0.3	Wittrock	Array operations
01/07/2002	0.4	Wittrock	"HALT" and "START" token description added, MGR_LOAD command and messages updated. New UDP Manager command for direct calls of UDPs with parameters
09/11/2002	0.5	Wittrock	"PMREAD" and "PMWRITE" tokens added
05/15/2003	1.0	Wittrock	Token compression description, message IDs updated; "PMSET" token added; Autostart mechanism added, new settings option: max. no. of messages per second.

1.3 Abbreviations

Table 2 Abbreviations

OCT	OSIRIS Command Token
RTL	Runtime Library
UDP	User Defined Procedure

2 Introduction

The OSIRIS Command Token Interpreter (OCT-interpreter) is used to execute user defined procedures (UDPs) on the spacecraft. Upload, execution and deletion of UDPs and settings of the OCT-interpreter are handled by the UDP-manager, which receives data via service 6. The UDP manager is used for creating timeline entries that start the execution of certain UDPs at given times, also.

3 Instruction Set

This chapter describes the instruction set of the OCT interpreter. It is strongly recommended to use the OCL compiler for token code creation. There should be no need to create token code by hand. Take this chapter as help for understanding the code that is generated by the compiler.

3.1 Code Format

All token interpreter commands has to fit the following format:

OPCODE	line number in source code	parameter 1	parameter 2	parameter 3
1 byte	3 bytes	4 bytes	4 bytes	4 bytes

The commands have all the same size of 16 bytes. If a command does not need all three possible parameters, the unused parameters may be set to any value (preferable 0). The corresponding line number is used for runtime messages only and should hold the number of the line in the source file that created this token command.

Any program (UDP) for the token interpreter is built like that:

INIT command	must	command for initialization of the activation record
pointers to array constants	opt.	the array constants are placed at the end of the code
pointers to static variables	opt.	static variables are placed behind the constants
pointers to externalized (large) variables	opt.	the first pointer has to be zero, all following pointers has to hold the offsets. The MEM command will add the start address of the dynamically allocated memory block to all entries.
pointers to externalized (large) help variables	opt.	large variables of known size are placed in the same dynamically allocated block as the externalized user variables. Because of that the pointer block just counts on and does not start with zero. The MEM command will set these pointers also.
constant values	opt.	block of simple type constants.
block handles	opt.	all block handle pointers are initialized with zero
zero filling space	must	this memory space is used to enlarge the data block to a multiple of the virtual machine command size

MEM command	opt.	if any externalized variables occur, this command initializes the according pointer blocks.
command block	opt.	user defined program...
RTS command	must	this command clears all local block handles and the externalized variables and returns to the calling code
array constants	opt.	data referenced by the 'pointers to array constants' block.
static memory	opt.	data referenced by the 'pointers to static variables' block

3.2 Activation Record

The command token parameters may refer to local memory addresses. These addresses are specified relative to the position of the local memory on the stack.

The following table shows the use of the relative addresses. Please note that the local addresses 1 and 2 **must** contain the return address and old stack base since these addresses are used implicitly by the **JSR**, **SJSR** and **RTS** token. The other addresses may be used for other purposes, but the OCL compiler creates code that follows the shown description.

Local Address	Entry	Use
...-1	function parameters	recommended
0	pointer to buffer for return value	recommended
1	return address	**must**
2	old stack base (position of former zero-index)	**must**
3...x	constant area, consisting of • 4 bytes reference to QLook buffer • references to array constants • references to static variables • references to externalized variables • constant values	optional
x...	variable area, consisting of • block handles • local variables • help variables	optional

3.3 Text Conventions

value	immediate (constant value)
&value	value on local memory, specified by a local memory index (value of "local address" in section 3.2)
&&value	value in external memory, specified by the local memory index of the absolute pointer to the external memory

3.4 Jumps

3.4.1 JSR (Jump to local subroutine)

JSR	distance of jump	-	-

The distance of the jump is measured in commands, not bytes. A jump of 0 will execute the same command and cause an endless loop.

3.4.2 SJSR (Jump to subroutine via symbol table)

SJSR	index in symbol table	size of parameters	size of return value

Jumps into a G-virtual machine code subroutine.

The destination of this jump is evaluated by a symbol table lookup. The read address is found at the given index in the global symbol table. The Program Counter will be set to the found address. A RTS command will restore the Program Counter to the next command directly behind the SJSR. Since SJSR does some registration operations to allow exchanging global subroutines at runtime (although this feature is not yet implemented), the next command must be BFGS to take the registrations back.

The jump will only be executed if the size of parameters (in long integer units) and the size of the return parameter given by the second and third parameter fit the sizes found in the symbol table.

3.4.3 CALL (Jump to external subroutine)

CALL	index in symbol table	size of parameter block	return type

Jumps into a runtime library function (real machine code).

The destination address of this jump is found in the symbol table at the given index. The size of the parameter block to use inside the called function is given in the second parameter. The parameter

block has to be placed at the top of the stack. The third parameter is a flag for the desired return type (0 means void, 1 double floating point and 2 long integer return value).

The jump will only be executed if the size of the parameter block (in long integer units) and the size of the return parameter given by the second and third parameter fit the sizes found in the symbol table.

3.4.4 JMP (Immediate jump)

JMP	distance of jump	-	-

The distance of the jump is measured in commands, not bytes. A jump of 0 will execute the same command and cause an endless loop.

3.4.5 JMPZ (Jump on zero)

JMPZ	distance on zero	distance otherwise	-

The distance of the jump is measured in commands, not bytes. A jump of 0 will execute the same command and cause an endless loop.

3.4.6 JMPN (Jump on negative)

JMPN	distance on negative	distance otherwise	-

The distance of the jump is measured in commands, not bytes. A jump of 0 will execute the same command and cause an endless loop.

3.4.7 INIT (Initialize activation record)

INIT	bytes of local data	number of bytes to copy for initialization	-

This command initializes the activation record of the subroutine on the local virtual machine stack. It reserves parameter1 bytes on the stack for local variables and constants and copies parameter2 bytes of the code following this command into the activation record. The program execution continues behind this data block. The data block has to be padded to fit a multiple of the command size.

3.4.8 RTS (Return and free external memory)

RTS	-	-	&allocated memory

Cleans up dynamically allocated memory and returns to the calling code. The third parameter is the address of the pointer to the dynamically allocated memory block (if it is 0, no memory is freed).

3.4.9 BFGS (Back from symbol table-subroutine)

BFGS	index of left subroutine	number of bytes on stack to free	ID of current function (for runtime messages)

This command has to be the next command after an SJSR command. It cancels registration actions and deletes the parameters of the left subroutine from the stack. The first parameter is used to mark the symbol table entry of the function as currently not in use (so it may be exchanged). The second parameter is the number of bytes that have been pushed to the stack as parameters for the function (the reference to the return value is excluded from this number). The third parameter is used for runtime messages only. It has to be the ID of the function that contains this BFGS command.

3.5 Memory

3.5.1 MEM (Allocate external memory and initialize pointers)

MEM	number of bytes	&pointer block	number of pointers

The MEM command dynamically allocates parameter1 bytes of memory and adds the address of the allocated memory to parameter3 offsets starting at parameter2.

3.5.2 ALLOC (Allocate temporary memory)

ALLOC	&number of elements	size of one element	&pointer to set

ALLOC is used to allocate temporary memory of sizes that are not known at compile time. Since most allocations are needed for array memory, the size is the product of the number of elements and the size of one element. The address of the allocated memory is stored in the variable specified by the third parameter.

3.5.3 FREE (Free allocated temporary memory)

FREE	&pointer	-	-

This is the opposite of ALLOC. FREE de-allocates the memory specified by the first parameter.

3.5.4 ADDR (Put address to memory)

ADDR	local memory index	&destination	-

This command is used to put the real address of a local memory position into local memory. The result may be used for external data accesses.

3.5.5 PUTVI (Put long integer constant to memory)

PUTVI	value	&destination	-

3.5.6 PUTVL (Put value to memory)

PUTVL	value	&destination	number of bytes

3.6 Stack Operations

3.6.1 PUSH (Push memory to stack)

PUSH	&memory	number of bytes	-

3.6.2 PUSHV (Push value to stack)

PUSHV	value	size of value (in bytes)	-

3.6.3 PUSHE (Push external memory to stack)

PUSHE	&&memory	number of bytes	-

3.6.4 PUSHI (Push memory with offset to stack)

PUSHI	&memory	&offset	number of bytes

3.6.5 PUSHA (Push address of local memory to stack)

PUSHA	&memory	-	-

3.6.6 PUSHIA (Push address of indicated local memory to stack)

PUSHIA	&[memory]	-	-

3.7 Flags

Since the OCT interpreter itself is linear executed software, flags setting cannot be done parallel (in background) and needs additional time. For better performance this is done only on request by the following commands.

3.7.1 FLGI (Set flags depending on integer)

FLGI	&integer	-	-

3.7.2 FLGF (Set flags depending on floating point)

FLGF	&float	-	-

3.7.3 TESTI (Check integer non-zero)

TESTI	&integer	&boolean	-

This command is used to convert an integer value into a boolean. All non-zero values are converted to true, only zero is converted to false.

3.7.4 TESTF (Check floating point non-zero)

TESTF	&float	&boolean	-

This command is used to convert a floating point value into a boolean. All non-zero values are converted to true, only zero is converted to false.

3.8 Range Checks

3.8.1 LIMIT (Limit integer to value)

LIMIT	&input	limit	&limited

This command is used to make sure that a value keeps in range. The result of the limitation of ***input*** to ***limit*** is stored in ***limited***. If ***input*** is out of range, an error message is produced. The OCL-compiler generates this command for array range checks.

3.8.2 LIMITI (Limit integer to integer)

LIMITI	&input	&limit	&limited

3.8.3 CHKFIT (Check equality with warning)

CHKFIT	&value1	&value2	&minimum

This command is used by the OCL-compiler to calculate the largest common size of two arrays. If ***value1*** and ***value2*** differ, an error message is produced.

3.8.4 VALID (Check immediate equality or greater with error)

VALID	&input	minimum	-

This command is used by the OCL-compiler to check the size of an array that is used as referenced parameter. If ***input*** is less than ***minimum***, an error message is generated and the execution aborted.

3.8.5 VALIDI (Check equality or greater with error)

VALIDI	&input	&minimum	-

3.9 Local Memory Read Access (with Type Conversion)

LDx	&source	&destination	-

3.9.1 LDI (Load local long with local long)

3.9.2 LDSF (Load local long with local float)

3.9.3 LDFS4 (Load local float with local signed long)

3.9.4 LDFF (Load local float with local float)

3.9.5 LDFU4 (Load local float with local unsigned long)

3.10 Program Memory Access

The UDP manager preserves 12800 * 32 bit of program memory for reliable storage of global UDP variables. Since the OCL token interpreter cannot do calculations inside of PM, the following two tokens are used to exchange values between program memory and data memory.

PM read/write accesses have built-in range checks to prevent illegal accesses.

3.10.1 PMREAD (Read from PM into DM)

PMREAD	&&source_offset	&&size	&&destination

The value of ***source_offset*** specifies the address of the first element which has to be copied relatively to the beginning of the preserved PM area. ***size*** specifies the number of elements to be copied and ***destination*** points to the destination inside data memory.

3.10.2 PMWRITE (Write from DM to PM)

PMWRITE	&&source	&&destination_offset	&&size

The value of ***source*** points to the source data memory. The ***destination_offset*** specifies the address relatively to the beginning of the preserved PM area where the first element has to be stored. ***size*** specifies the number of elements to be copied.

3.10.3 PMSET (Fill PM area with value)

PMSET	&&destination_offset	&&value	&&size

The ***destination_offset*** specifies the address relatively to the beginning of the preserved PM area where the first element has to be stored. The next ***size*** elements inside the PM are set to ***value***.

3.11 External Memory Read Access (with Type Conversion)

ELDx	&&source	&destination	-

3.11.1 ELDI (Load local long with external long)

3.11.2 ELDSF (Load local long with external float)

3.11.3 ELDFS4 (Load local float with external signed long)

3.11.4 ELDFF (Load local float with external float)

3.11.5 ELDFU4 (Load local float with external unsigned long)

3.12 Local Memory Write Access (with Type Conversion)

STRx	&source	&destination	-

3.12.1 STRI4 (Store local long into local long)

3.12.2 STRSF (Store local signed long into local float)

3.12.3 STRUF (Store local unsigned long into local float)

3.12.4 STRFS (Store local float into local signed long)

3.12.5 STRFU (Store local float into local unsigned long)

3.12.6 STRFF (Store local float into local float)

3.13 External Memory Write Access (with Type Conversion)

ESTRx	&source	&&destination	-

3.13.1 ESTRI4 (Store local long into external long)

3.13.2 ESTRSF (Store local signed long into external float)

3.13.3 ESTRUF (Store local unsigned long into external float)

3.13.4 ESTRFS (Store local float into external signed long)

3.13.5 ESTRFU (Store local float into external unsigned long)

3.13.6 ESTRFF (Store local float into external float)

3.14 Copy Commands

3.14.1 CPY (Copy a number of bytes inside local memory)

3.14.2 ICPY (Copy a number of bytes inside indexed local memory)

3.14.3 ECPY (Copy a number of bytes inside external memory)

3.14.4 CPYL (Copy a fix number of bytes inside local memory)

3.14.5 ICPYL (Copy a fix number of bytes inside indexed local memory)

3.14.6 ECPYL (Copy a fix number of bytes inside external memory)

3.14.7 CPYED (Copy a fix number of bytes from external to local memory)

3.14.8 CPYDE (Copy a fix number of bytes from local to external memory)

3.14.9 CPYEI (Copy a fix number of bytes from ext. to indexed loc. mem)

3.14.10 CPYIE (Copy a fix number of bytes from indexed loc. to ext. mem)

3.15 Memory Fill

3.15.1 FILLV (Fill a fix number of local bytes with a fix value)

3.15.2 EFILLV (Fill a fix number of external bytes with a fix value)

3.15.3 EFILVL (Fill a number of external bytes with a fix value)

3.16 Array Operations

Array operations consist of a sequence of three commands:

1. Set the operation to execute on elements (**`SET?OP`**)
2. Set the configuration of the element counters (**`SETCNT`**)
3. Execute the operation with this configuration (**`OP??`**)

This allows direct operations on different input types.

3.16.1 SETFOP (Set floating point operation)

SETFOP	operation ID	-	-

Specify the operation to execute on the following floating point OPxx command. The operation ID can be one of the following:

0	addition
1	subtraction
2	negation
3	multiplication
4	division
5	minimum
6	maximum
7	power
8	absolute value
9	sine
10	arc sine

11	cosine
12	arc cosine
13	tangent
14	arc tangent
15	natural logarithm
16	e^x
17	logarithm (base 10)
18	random
19	is equal
20	is less

3.16.2 SETIOP (Set integer operation)

SETIOP	operation ID	-	-

Specify the operation to execute on the following integer OPxx command. The operation ID can be one of the following:

ID	signed/ unsigned	
0	u	addition
1	s	
2	u	subtraction
3	s	
4	u	negation
5	s	
6	u	multiplication
7	s	
8	u	division
9	s	
10	u	modulo
11	s	
12	u	minimum
13	s	
14	u	maximum
15	s	
16	u	shift left

17	s	
18	u	shift right
19	s	
20	u	power
21	s	
22	u	random
23	s	
24	u	and
25	s	
26	u	or
27	s	
28	u	xor
29	s	
30	u	not
31	s	
32		abs
33		is equal
34	u	is less
35	s	

3.16.3 SETPOP (Set put operation)

SETPOP	operation ID	-	-

Specify the operation to execute on the following OPPUT command. The following IDs are allowed to use for array type conversions:

1	put floating point to floating point
2	put unsigned integer to floating point
3	put signed integer to floating point
4	put floating point to signed integer
5	put floating point to boolean
6	put integer to boolean

3.16.4 SETCNT (Set counter configuration)

SETCNT	increment of first operand address	increment of second operand address	number of elements to process

Configure the counters for the iteration. If the operand is an array, the increment of the operand address has to be the size of a single array element. If the operand is a single value the increment of the address has to be zero (so all iterations will operate on the same value). This makes it possible to execute operations on all combinations of scalar values and arrays. Since the result is always an array with a known element size, the user can not define the destination increment. The third parameter specifies the number of iterations.

3.16.5 OPFF (Operate on floating point and floating point)

OPFF	&&start of first operand	&&start of second operand	&&start of result

Iterate the operation specified by SETFOP on first and second floating point operand and put the result in an array of floating point values. After each iteration, the addresses of the first and second operand are incremented according to the configuration through SETCNT before and the address of the result is incremented by one element also.

3.16.6 OPFUL (Operate on floating point and unsigned long)

Like OPFF (3.16.5), but iterate the operation specified by SETFOP on first floating point and second unsigned long operand and put the result in an array of floating point values. The unsigned long operand is converted into floating point before the operation.

3.16.7 OPULF (Operate on unsigned long and floating point)

Like OPFUL (3.16.6), but the operand types are swapped.

3.16.8 OPFSL (Operate on floating point and signed long)

Like OPFUL (3.16.6), but the second operand is supposed to be signed long.

3.16.9 OPSLF (Operate on signed long and floating point)

Like OPFSL (3.16.8), but the operand types are swapped.

3.16.10 OPXLXL (Operate on long and long)

OPXLXL	&&start of first operand	&&start of second operand	&&start of result

Iterate the operation specified by SETIOP on first and second floating point operand and put the result in an array of long integer values. After each iteration, the addresses of the first and second operand are incremented according to the configuration through SETCNT before and the address of the result is incremented by one element.

3.16.11 OPFFB (Operate on float and float to boolean)

OPFFB	&&start of first operand	&&start of second operand	&&start of result

Iterate the operation specified by SETFOP on first and second floating point operand and put the result in an array of boolean values.

3.16.12 OPFULB (Operate on float and unsigned long to boolean)

OPFULB	&&start of first operand	&&start of second operand	&&start of result

3.16.13 OPULFB (Operate on unsigned long and float to boolean)

OPULFB	&&start of first operand	&&start of second operand	&&start of result

3.16.14 OPFSLB (Operate on float and signed long to boolean)

OPFSLB	&&start of first operand	&&start of second operand	&&start of result

3.16.15 OPSSLFB (Operate on signed long and float to boolean)

OPSLFB	&&start of first operand	&&start of second operand	&&start of result

3.16.16 OPPUT (Put with array conversion)

OPPUT	&&start of source	&&start of result	

Iterate the operation specified by SETPOP on the source operand and put it to the result. Both addresses are incremented by the sizes specified by SETCNT after each iteration.

3.17 Integer Calculations

TOKEN	first operand	second operand	result

3.17.1 ADDS (Add two longs)

3.17.2 ADDVS (Add a constant and a long)

3.17.3 SUBS (Subtract two longs)

3.17.4 SUBVS (Subtract a long from a constant)

3.17.5 NEGS (Negate a long)

3.17.6 MULTS (Multiply two signed longs)

3.17.7 MULTVS (Multiply a constant and a signed long)

3.17.8 DIVS (Divide two signed longs)

3.17.9 DIVVS (Divide a constant by a signed long)

3.17.10 DIVSV (Divide a signed long by a constant)

3.17.11 MODS (Derive the modulo of the division of two signed longs)

3.17.12 MODVS (Derive the modulo of the division of a const and a signed long)

3.17.13 MODSV (Derive the modulo of the division of a signed long and a const)

3.17.14 MINS (Evaluate the minimum of two signed longs)

3.17.15 MINVS (Evaluate the minimum of a constant and a signed long)

3.17.16 MAXS (Evaluate the maximum of two signed longs)

3.17.17 MAXVS (Evaluate the maximum of a constant and a signed long)

3.17.18 SHLS (Shift left of a signed long by a constant)

3.17.19 SHRS (Shift right of a signed long by a constant)

3.17.20 PWRS (Power a signed long by another signed long)

3.17.21 DIVU (Divide two unsigned longs)

3.17.22 DIVVU (Divide a constant by an unsigned long)

3.17.23 DIVUV (Divide an unsigned long by a constant)

3.17.24 MODU (Derive the modulo of a constant and an unsigned long)

3.17.25 MODVU (Derive the modulo of a constant and an unsigned long)

3.17.26 MODUV (Derive the modulo of an unsigned long and a constant)

3.17.27 MINU (Evaluate the minimum of two unsigned longs)

3.17.28 MAXU (Evaluate the maximum of two unsigned longs)

3.17.29 MINVU (Evaluate the minimum of a constant and an unsigned long)

3.17.30 MAXVU (Evaluate the maximum of a constant and an unsigned long)

3.17.31 SHLU (Shift left of an unsigned long by a constant)

3.17.32 SHRU (Shift left of an unsigned long by a constant)

3.17.33 PWRU (Shift left of an unsigned long by a constant)

3.18 Floating Point Calculations

3.18.1 ADDF (Add two floats)

3.18.2 SUBF (Subtract two floats)

3.18.3 NEGF (Negate a float)

3.18.4 MULTF (Multiply two floats)

3.18.5 DIVF (Divide two floats)

3.18.6 MINF (Derive the minimum of two floats)

3.18.7 MAXF (Derive the maximum of two floats)

3.18.8 PWRF (Power two floats)

3.18.9 SINF (Sine of a float)

3.18.10 ASINF (Arcsine of a float)

3.18.11 COSF (Cosine of a float)

3.18.12 ACOSF (Arccosine of a float)

3.18.13 TANF (Tangent of a float)

3.18.14 ATANF (Arctangent of a float)

3.18.15 LOGF (Logarithm of a float)

3.18.16 EXPF (Power e by a float)

3.19 Random Operations

3.19.1 SEED (Set a start value for the random number generator)

3.19.2 RDMZ (Randomize the start value of the random number generator)

3.19.3 RAND (Calculate a random number)

3.20 Binary Operations

3.20.1 AND (Binary AND of two longs)

3.20.2 ANDV (Binary AND of a constant value and a long)

3.20.3 OR (Binary OR of two longs)

3.20.4 ORV (Binary OR of a constant value and a long)

3.20.5 XOR (Binary XOR of two longs)

3.20.6 XORV (Binary XOR of a constant value and a long)

3.20.7 NOT (Binary NOT of a long)

3.20.8 NOTB (Boolean NOT of a long)

3.21 Comparison

3.21.1 EQI4 (Check equality of two longs)

3.21.2 EQVI4 (Check equality of a constant value and a long)

3.21.3 EQF (Check equality of two floats)

3.21.4 EQE (Check equality of two external areas)

EQE	&&source	&&destination	&length / &result

EQE compares the memory areas specified by the first and second parameter. The command reads out the local memory specified by the third parameter to determine the size of the areas to compare and it will overwrite this memory with the result of the comparison.

3.21.5 SLS (Check signed long is less than signed long)

3.21.6 SLSV (Check signed constant is less than signed long)

3.21.7 SGTV (Check signed constant is greater than signed long)

3.21.8 LSF (Check float is less than signed long)

3.21.9 ULS (Check unsigned long is less than unsigned long)

3.21.10 ULSV (Check unsigned constant is less than unsigned long)

3.21.11 UGTV (Check unsigned constant is greater than unsigned long)

3.21.12 LSE (Check external area is less than another)

LSE	&&source	&&destination	&length / &result

LSE compares the memory areas specified by the first and second parameter. The command reads out the local memory specified by the third parameter to determine the size of the areas to compare and it will overwrite this memory with the result of the comparison.

3.22 Miscellaneous

3.22.1 WTEV (Wait for event or block)

WTEV	&event	&timeout	&result

WTEV stops the execution of the current UDP until the event is received or the timeout is reached. During this time the token interpreter does not react on debugging commands like **MGR_SUSPEND** or **MGR_QUIT** (see 4.3.8 - 4.3.11). **result** is the result of the called Virtuoso function **KS_EventTestWT**.

3.22.2 SIGNAL (Set event)

SIGNAL	&event	-	-

SIGNAL sets the specified event (see 3.22.1, **WTEV**).

3.22.3 SLEEP (Sleep a specified time)

SLEEP	&ticks	-	-

The execution of the current UDP is stopped for the specified number of time-ticks. During this time the token interpreter does not react on debugging commands like **MGR_SUSPEND** or **MGR_QUIT** (see 4.3.8 - 4.3.11).

3.22.4 AT (Wait until a specified time)

Not yet implemented.

3.22.5 HALT (Pause execution)

HALT	-	-	-

HALT causes the token interpreter to stop the execution of the current UDP immediately. The execution can be continued by sending a continue signal (see 4.3.9). It is also possible to continue the execution token by token (see 4.3.10).

3.22.6 START (Start the execution of a UDP by another token interpreter)

START	ID	&name	-

START sends a signal to the UDP manager to start the execution of the specified UDP by another token interpreter in a different task. Before sending this signal, the name of the UDP currently stored at the given position of the symbol table is compared to the name specified by this command to ensure that it is in fact the desired UDP (the UDP itself may be replaced, but as long as the name of the UDP remains the same, it's functionality will most probably be correct).

3.22.7 NOP (Do nothing)

NOP	-	-	-

4 UDP Manager

The UDP manager receives command blocks (via service 6 subtype 2 with ID 114) and processes their data depending on the first value of the block (i.e. command-ID). It is possible to send a whole list of command blocks at once. Normally, you should use the output file of the OCL compiler as input for the UDP manager.

4.1 UDP IDs

The UDP Manager stores a table of all UDPs. The first entries of this table (and therefore even the lower IDs) are used by the built-in runtime library functions. The next 10 entries are preserved for temporary UDPs that are deleted automatically after execution. All following IDs up to the end of the table are available for normal UDPs.

4.2 Startup Procedure

The UDP Manager loads on startup the UDP with the lowest possible ID (= number of built-in functions + 10) from NVRAM. If this UDP is called “AutoStart” (case sensitive!) and does neither expect any arguments nor returns any value, that UDP is executed automatically. This UDP has to do all other desired startup actions (e.g. load other POPs and start demon UDPs).

4.3 Input

4.3.1 Single UDP Upload for Later Execution

Load a UDP into spacecraft memory for later use.

0x00030000	(MGR_SINGLELOAD)
long	UDP ID
long	Time stamp (creation time)
32 characters	UDP name
long	size of provided parameters (in long units)
long	size of returned value (in long units)
long	number of uses before automatic remove from spacecraft memory (-1: do not delete automatically)
long	size of token code block
long	number of pointers that has to be initialized (position of static

	variables depend on UDP position in S/C memory)
x * 4 longs	UDP code
0x54495749	synchronization mark

If the sync mark is not found at the correct place, the UDP upload is rejected.

4.3.2 Single UDP Upload with Immediate Execution

0x00030002	(MGR_EXECLOAD)
…	like single UDP upload for later execution (starting with UDP ID)

Like single UDP load, but the uploaded UDP will be executed immediately.

4.3.3 Multiple UDP Upload

The Multi-UDP-Upload command is used for different purposes: For upload of uncompressed token code, for upload and decompression of compressed token code and for uploading or deleting user defined decompression tables. The desired action is specified by the two uppermost bits of the long integer value directly behind the UDP-Manager command ID (MGR_MULTILOAD).

4.3.3.1 Upload of uncompressed UDP code

0x00030001	(MGR_MULTILOAD)
long	Number of UDPs in this block (<= 0x3fffffff)
…	for each UDP a block like described for single UDP upload (starting with UDP ID)

4.3.3.2 Upload of compressed UDP code

0x00030001	(MGR_MULTILOAD)
long	Number of UDPs in this block (<= 0x3fffffff) | 0x80000000
long	Magic number of used compression table
…	Compressed UDP token code. The uncompressed data format is the same as described for single UDP upload.

All UDPs are compressed as one data block. The compression uses a compression table which is not part of the uploaded data block. The corresponding decompression table has to be present on board already.

4.3.3.3 Upload of new decompression table

Although the built-in decompression table allows a good compression level for current UDP token code, future uploads might want to use another compression table for better compression results.

A new decompression table can be uploaded using the following command:

0x00030001	(MGR_MULTILOAD)
long	Size of table (in 32-bit units) | 0x40000000
…	Decompression table data
long	Magic number of compression table
long	Checksum of decompression table and it's magic number

Please note that the uploaded decompression table will be used for all further compressed uploads, unless a UDP manager reset command is processed or the table is successfully replaced by a new one.

For further information about the decompression table format see document IDA-OCL-0004.

To remove the user defined table and turn back to the built-in table, use the following command:

0x00030001	(MGR_MULTILOAD)
long	0x40000000

4.3.4 Single UDP Remove

0x00000005	(MGR_DELETE)
long	ID of UDP
long	timestamp of UDP

4.3.5 Single UDP Replace

0x00030006	(MGR_RELOAD)
long	timestamp of old UDP
…	like single UDP upload for later execution (starting with UDP ID)

For successful replacing, the given timestamp of the old UDP has to fit the timestamp of the according UDP present at spacecraft. The UDP name, size of return value and the size of parameters

have to fit the parameters present in the spacecraft's symbol table, also. If any difference is found, the replacement will be rejected.

4.3.6 Make Timeline Entry

0x00000004	(MGR_TIMELINE)
long	system time in seconds to start
long	fractions of time to start
long	ID of UDP to start at given time

4.3.7 Execute UDP

0x00000003	(MGR_EXEC)
long	ID of UDP to execute

4.3.8 Stop UDP execution

0x00000007	(MGR_SUSPEND)
long	ID of UDP to suspend

4.3.9 Continue UDP execution

0x00000008	(MGR_RESUME)
long	ID of UDP to resume

4.3.10 Execute Next UDP Token

0x00000009	(MGR_STEP)
long	ID of UDP to step through

4.3.11 Quit UDP execution

0x0000000A	(MGR_QUIT)
long	ID of UDP to quit

4.3.12 Reset UDP Manager

0x0000000B	(MGR_RESTART)

0xFFFFFFF4	(~MGR_RESTART)
0x54495749	synchronization mark

4.3.13 Settings

0x0000000C	(MGR_SETTINGS)
long	maximum stack size (default: 1024)
long	maximum number of errors before aborting the UDP execution (default: 10)
long	symbol table size (maximum UDP ID, default: 768)
long	timeline size (maximum number of entries, default: 64)
long	time tolerance (milliseconds) for delayed UDP start from internal timeline. If the desired start time expired by more than this amount, the UDP will not be started (default: 1920).
long	Verbosity mode value: 0: quiet mode, not even error messages are generated 1: terse, generate only error messages 5: normal, suppress mass-messages (e.g. don't show messages for each loaded UDP during multiple UDP upload) 10: verbose, currently there is no message which would be suppressed in this mode. It might be useful for debugging messages… 20: babble, show all messages. The default mode value is 5. It will be restored after a UDP Manager reset.
long	Maximum number of messages per second (OCL system will keep own message rate below this limit by delaying execution if necessary). The default value is 15.
0x54495749	synchronization mark

Settings are not changed, if sync mark is not found at the expected place. If any of these values in the TC packet is set to -1, the current value will not be changed (e.g. sending a "0x0000000c -1 -1 -1 -1 -1 10 12" command will set only the verbosity mode to 'verbose' and limits the number of messages per second to 12.

4.3.14 Save

0x0000000D	(MGR_SAVE)
long	ID of first UDP to save to NVRAM
long	(ID of last UDP to save to NVRAM) + 1

All UDPs from the first specified to the last are saved to NVRAM.

If the first parameter refers to a built-in function, it is set to the first UDP ID. If the second parameter exceeds the number of UDPs, it is clipped. If you want to save all UDPs you may set the first parameter to 0 and the second to –1 (0xffffffff).

If the second parameter exceeds the highest ID of currently available UDPs or is set to -1, an EOF marker is written behind the last saved UDP. Possibly existing POPs behind this marker will not be accessible any more.

If the second parameter is at most equal to the highest ID of all currently available UDPs, the EOF marker is not written. In this case, some older POPs may be overwritten, but all untouched former POPs remain usable.

4.3.15 Load

0x0000000E	(MGR_LOAD)
long	ID of UDP to be loaded

The specified UDP is loaded from NVRAM into DPU memory. To load all UDPs available in NVRAM at once, specify –1 (0xffffffff).

If the ID specifies an internal UDP (preferable 0), the current state of the NVRAM is checked and returned via housekeeping.

4.3.16 Execute UDP with Parameters

0x0000000F	(MGR_PARAMCALL)
32 characters	Name of UDP to be executed
long	User-tag of this call
long	Number of following long integers parameter data
…	Parameter data

This command generates a temporary UDP that consists only of a call to the UDP with the specified name. The name of the generated UDP is built by the four characters "`run `" followed by the name

of the called UDP. To differentiate messages returned by various calls to the same UDP, the "user-tag" field of this command should contain a unique ID for each different call to the same UDP.

Important note: Use this command for calls to UDPs with plain parameters only. It is not allowed to use this command to call UDPs that have reference parameters (declared with **&**). Misuse will cause **OCL system crashes**!

Be aware that this command does not allow type checking of the UDP parameters. It also does not allow calling built-in (runtime library) functions.

The UDP Manager looks for the specified UDP name in the on-board symbol table. If the UDP is found and the over-all size of the expected parameters fits the size of the specified parameter data, the UDP Manager generates the temporary UDP with the according call to the found UDP. The generated UDP gets a currently unused ID from the pool of IDs that are preserved for temporary UDPs. The temporary UDP will be executed immediately and is deleted automatically after execution.

4.4 Output

The DPU based UDP manager sends messages via telemetry data with event ID 0xA9F6 (43510) up to 0xAA2B (43563). The event ID is repeated with the next short value, which is followed by further information in variable format. The length of returned information may vary but will not exceed 100 bytes.

4.4.1 Informational Messages

4.4.1.1 SUSPENDED (0xA9F6)

The execution of the specified UDP has been suspended at the mentioned token interpreter.

Additional information is formatted like described in 4.4.4.1. Furthermore, the current settings of the run-state flags are returned inside the additional long:

- 0x01 – UDP is suspended
- 0x02 – UDP will abort on next token fetch
- 0x04 – UDP is executed in step mode
- if none of these bits is set, the UDP will be executed in normal mode

4.4.1.2 RUNNING (0xA9F7)

The given UDP has been started successfully on the specified token interpreter.

Additional information is formatted like described in 4.4.4.1.

4.4.1.3 FINISHED_NORM (0xA9F8)

The execution of the given UDP has been finished normally (without errors).

Additional information is formatted like described in 4.4.4.1.

4.4.1.4 UDP_LOADED (0xA9F9)

The given UDP has been successfully loaded into DPU memory. Additional information is formatted like described in 4.4.4.2.

4.4.1.5 UDP_DELETED (0xA9FA)

The UDP has been successfully deleted from DPU. Additional information is formatted like described in 4.4.4.2.

4.4.1.6 SETTINGS_SET (0xA9FB)

The settings of UDP manager and token interpreter have been set to the returned values. Additional information is formatted like described in 4.4.4.3.

4.4.1.7 MANAGER_RESET (0xA9FC)

The UDP manager has been reset. The message-ID is followed by a long integer value that contains the number of UDPs that could not have been deleted.

4.4.1.8 POP_LOADED (0xA9FD)

The specified POP has been loaded successfully from EEPROM into DPU memory. Additional information is formatted like described in 4.4.4.2.

4.4.1.9 POP_SAVED (0xA9FE)

The specified UDP/POP has been stored successfully into EEPROM. Additional information is formatted like described in 4.4.4.2.

4.4.1.10 POP_INFO (0xA9FF)

This message contains header information about one POP, like described in 4.4.4.2, with some additional values:

1 long	Address in NVRAM
1 long	Unused
1 long	State of POP (0: okay, 1: not checked, 2: failed, 3: zero length, 4: header corrupt, 5: sync error)

The state “not checked” means that the POP body could not be checked due to very low memory. The POP may be stored correctly but it may be corrupted as well. This check does not compare the

version in the NVRAM and the corresponding in the RAM, but checks the overall checksum of the POP only.

Please note that the information from the POP header may be invalid, if the state flag is 4 or 5.

4.4.1.11 POP_DONE (0xAA00)

This message specifies the end of POP_INFO messages. It contains the size of the used NVRAM and the summary of check results.

1 long	Size of used NVRAM space
1 long	Number of successfully checked UDPs
1 long	Number of warnings during check
1 long	Number of failed UDP checks

4.4.1.12 MULTI_LOAD (0xAA01)

This message contains summary information about multiple UDP upload.

1 long	Number of successfully loaded UDPs
1 long	Number of warnings during upload
1 long	Number of failed UDP uploads

4.4.1.13 MULTI_POP_LOAD (0xAA02)

This message contains summary information about the completion of POP to UDP loading or POP checking.

1 long	Number of successfully loaded POPs or Number of successfully checked POPs
1 long	Number of warnings during upload / check
1 long	Number of errors

4.4.1.14 MULTI_POP_SAVE (0xAA03)

This message contains summary information about the completion of storing UDPs as POPs into NVRAM..

1 long	Number of successfully stored POPs

1 long	Number of warnings during writing
1 long	Number of errors

4.4.2 Error Messages

4.4.2.1 FINISHED_ERR (0xAA09)

The specified UDP execution has been finished, but had runtime errors.

Additional information is formatted like described in 4.4.4.1. Furthermore, the type of errors that occurred during execution is returned:

- 0x0001 – stack overflow
- 0x0002 – array limits exceeded
- 0x0004 – zero handle
- 0x0008 – division by zero
- 0x0010 – not implemented
- 0x0020 – pure virtual function called
- 0x0040 – something went wrong inside extern function
- 0x0080 – symbol table inconsistent
- 0x0100 – too few parameters
- 0x0200 – memory error
- 0x0400 – internal error

4.4.2.2 ALREADY_DECL (0xAA0B)

The specified UDP ID is already in use. This can be caused by a repeated upload of the same UDP sequence or by an inconsistency between the ground based symbol table and its DPU based counterpart. Additional information is formatted like described in 4.4.4.2.

4.4.2.3 ARRAY_EXCEED (0xAA0C)

The token interpreter found an access that exceeded the array borders inside the specified UDP.

Additional information is formatted like described in 4.4.4.1. Furthermore, additional information about the error handling is given:

- 0 – aborting UDP execution (unrecoverable error)
- 1 – taking maximum value instead the faulty one
- 2 – taking minimum value instead
- 3 – taking zero instead
- 4 – ignoring operation

4.4.2.4 DIFF_DECL (0xAA0D)

The replacement command failed due to changes in the interface. UDP replacement is only possible if at least the interface remains unchanged. Additional information is formatted like described in 4.4.4.2.

4.4.2.5 DIV_BY_ZERO (0xAA0E)

A division by zero has been detected by the token interpreter.

Additional information is formatted like described in 4.4.4.1. Furthermore, information about error handling is given like described in ARRAY_EXCEED (0xAA0C), section 4.4.2.3.

4.4.2.6 ILLEGAL_VALUE (0xAA0F)

The value/ID is out of range. This error may occur if the given UDP ID is higher than the current size of symbol table. You should enlarge the table. Additional information is formatted like described in 4.4.4.2.

4.4.2.7 IN_USE (0xAA10)

The specified UDP could not have been deleted, since it is currently executed by at least one token interpreter.

Additional information is formatted like described in 4.4.4.2.

4.4.2.8 MAX_ERROR (0xAA11)

The maximum number of errors has been reached, the execution has been aborted by the given token interpreter.

Additional information is formatted like described in 4.4.4.1. Furthermore, the number of errors is returned as long value.

4.4.2.9 MEM_ERROR (0xAA12)

Not enough memory to complete the action. Additional information is formatted like described in 4.4.4.1.

4.4.2.10 PURE_VIRTUAL (0xAA13)

The token interpreter detected a call to a non-existing UDP. Probably the needed UDP has not been uploaded yet or has been deleted already. Perhaps there is another inconsistency between the ground based symbol table and its DPU based counterpart.

Additional information is formatted like described in 4.4.4.1, followed by the ID of the non-existing UDP.

4.4.2.11 SIZE_EXCEEDING (0xAA14)

The time line table is too small to hold all desired entries or the symbol table is too small to load all UDPs from NVRAM. This error may also occur if you try to shrink any table below the currently used size.

Additional information may contain the following data after a failed resizing:

- 1 long – desired table size
- 1 long – current table fill state

or after trying to add a new time line entry:

- 0xffffffff
- 1 long – current table fill state

4.4.2.12 STACK_OVERFLOW (0xAA15)

The stack of the given token interpreter is too small to hold all local variables. You should increase the stack size.

Additional information is formatted like described in 4.4.4.1. Furthermore the amount of exceeding longs is returned.

4.4.2.13 SYNC_ERROR (0xAA16)

The upload failed because the data seems to be corrupted. Additional information is formatted like described in 4.4.4.2.

4.4.2.14 TIMESTAMP_DIFFERS (0xAA17)

a) The desired action failed because the timestamp of the UDP currently loaded into DPU differs from that supposed by the ground based UDP manager. This may be caused by a repetition of an outdated UDP replacement command (or similar) or by an inconsistency of both parts of the symbol table. Additional information is formatted like described in 4.4.4.2.

b) The compressed upload failed, because the compression table used on ground does not correspond with the active decompression table on board. The message-ID is followed by the table's magic number during compression and the magic number of the active decompression table.

4.4.2.15 TOO_FEW (0xAA18)

The UDP that you've tried to start (using the MGR_EXEC command, see 4.3.7) expects parameters. UDPs of this kind can only be called by other UDPs and cannot be started directly.

Additional information is formatted like described in 4.4.4.1, followed by the number of missing parameters.

4.4.2.16 ZERO_HANDLE (0xAA19)

The UDP with the desired ID currently does not exist. It is not possible to start this UDP at the moment. The error may be caused by a MGR_EXEC command or by time line. The ID of the non-existing UDP is returned.

4.4.2.17 INTERPRETER_BUSY (0xAA1A)

All token interpreters are currently busy. The desired UDP could not have been started. Additional information is formatted like described in 4.4.4.2.

4.4.2.18 OUT_OF_DATE (0xAA1B)

A time line entry could not have been executed because it is out of date. This may occur if the DPU time has been changed. As additional information the message-ID is followed by

- 1 long integer that contains the ID of the UDP that should have been started,
- 2 long integers that contain the desired start time in seconds and fractions (1 long each)
- 2 long integers that contain the current system time in the same format

4.4.2.19 UNKNOWN_CMD (0xAA1C)

The UDP manager could not understand the command. The value of the unknown command is returned as long integer after the message-ID.

4.4.2.20 CHECKSUM (0xAA1D)

a) The checksum of the NVRAM data is incorrect. The message-ID is followed by the ID of the POP / UDP currently processed in NVRAM.

b) The checksum of the decompression table is incorrect. The message ID is followed by 0xFFFFFFFF, the required and the actual checksum.

4.4.2.21 START_FAILED (0xAA1E)

The desired UDP could not be started by a token interpreter in another thread. Additional information is formatted like described in 4.4.4.1, followed by the reason for the error:

- PURE_VIRTUAL: The desired UDP does not exist.
- DIFF_DECL: The interface of the called UDP does not fit.
- INTERPRETER_BUSY: Currently no interpreter thread available.

4.4.3 Fatal Error Messages

4.4.3.1 NOT_IMPLEMENTED (0xAA28)

The token interpreter found an unknown token. This indicates a corrupted UDP. Additional information is formatted like described in 4.4.4.1.

4.4.3.2 STACK_CORRUPTED (0xAA29)

Something went wrong during token interpretation. Additional information is formatted like described in 4.4.4.1.

4.4.3.3 SYMTAB_INCONSIST (0xAA2A)

The parameter and return value sizes expected by the called UDP / POP / built-in function differ from the data pushed by the calling UDP. Additional information is formatted like described in 4.4.4.1.

4.4.3.4 POP_INCONSIST (0xAA2B)

Something went wrong with NVRAM memory.

4.4.4 Additional Information

4.4.4.1 Standard Interpreter Created Information Block

32 bytes	global UDP name
1 long	current UDP ID
32 bytes	current UDP name
1 long	time stamp of current UDP
1 long	current call stack depth
1 long	corresponding source code line
1 long	address of current token
1 long	ID of token interpreter
1 long	additional information

4.4.4.2 Symbol Table Information Block

1 long	UDP ID
1 long	time stamp of this UDP
32 bytes	UDP name
1 long	size of parameters (in longs)
1 long	size of return value (in long)

4.4.4.3 Settings Information Block

1 long	stack size
1 long	maximum number of errors before abort
1 long	maximum number of entries in symbol table
1 long	maximum number of entries in time line
1 long	time tolerance in milliseconds
1 long	verbosity level
1 long	maximum number of messages per second