

<System_Name> API and Developers' Guide

- Stephen McGregor wrote this document from scratch and in its entirety.
- This document is only for private evaluation.
- Do not retain this document after use.
- Do NOT distribute this document.

This document is only for use in association with hiring Stephen McGregor as a technical writer.

Author: Stephen McGregor

Status: Complete final draft for review

Tuesday, February, 20XX

Version: 0.9.1

Table of Contents

1	INTRODUCTION.....	29
1.1	What is <System_Name>.....	29
1.2	Diagram: <System_Name> Software Structure.....	31
2	PROGRAMS DESCRIBED – BRIEF OVERVIEW.....	32
2.1	Input Programs.....	32
2.1.1	Normal Paging Input Programs.....	32
2.1.1.1	skymds00.....	32
2.1.1.2	skymds01.....	32
2.1.1.3	skymds01 maintenance: skymds01a to skymds01p.....	32
2.1.1.4	skymds08.....	32
2.1.2	Email / web to Pager, SMS and Bulk SMS messages.....	32
2.1.2.1	mdsemgw.....	32
2.1.2.2	mdsem01.....	32
2.1.2.3	mdsem02.....	32
2.1.2.4	mdsem03.....	32
2.1.2.5	mdsms01.....	32
2.1.3	From <System_Name> II (web-based) to <System_Name>.....	32
2.1.3.1	skymds96a.....	32
2.1.4	Direct Connection to <System_Name>.....	32
2.1.4.1	skymds02.....	32
2.1.4.2	skymds70.....	32
2.1.4.3	skymds18.....	32
2.2	Central processing and Maintenance programs.....	33
2.2.1	The main data repositories.....	33
2.2.1.1	The Bulk Queue.....	33
2.2.1.2	The Distribution Queues.....	33
2.2.1.3	The history Server, History Data.....	33
2.2.1.4	CDR – Call Detail Records.....	33
2.2.2	Maintenance of the <System_Name> System.....	33
2.2.2.1	skymds00j.....	33
2.2.2.2	<System_Name> “file” menu programs.....	33
2.2.2.3	<System_Name> “Report” menu programs.....	33
2.2.2.4	<System_Name> “customer” menu programs.....	33
2.2.2.5	<System_Name> “Utilities” menu programs.....	33
2.2.3	Processing Applications.....	33
2.2.3.1	skymds20.....	33
2.2.3.2	mdsalarm.....	33
2.2.4	Data replication to all <System_Name> sites – Maintenance queue.....	33
2.2.4.1	skymds23.....	33
2.2.4.2	skymds24.....	33
2.2.4.3	skymds28.....	33
2.3	Output / Delivery programs.....	34
2.3.1	The Destination systems.....	34
2.3.2	Delivery to external systems: skymds03.....	34
2.3.2.1	skymds03 sub-modules.....	34
2.3.3	Other external Delivery programs.....	34
2.3.3.1	skymds25.....	34
2.3.3.2	skymds10.....	34
2.3.3.3	skymds11t.....	34
2.3.4	In through the out-door: Input systems that co-operate with Output applications.....	34
2.3.4.1	skymds27.....	34
2.3.4.2	skymds06.....	34
2.3.4.3	skymds11r.....	34
2.4	Diagram: <System_Name> Hardware Structure.....	35

3	SKYMD00.....	38
3.1	skymds00 Overview.....	38
3.2	Running skymds00.....	38
3.2.1	Command line parameters.....	38
3.2.2	Example command-line.....	38
3.3	Program Description.....	38
3.3.1.1	Login window.....	38
3.3.1.2	skymds01 – take incoming calls.....	38
3.3.1.3	base menu.....	38
3.3.1.4	Submenus.....	39
3.4	GUI: Sample “SCREEN DUMPS”.....	39
3.4.1.1	Login screen.....	39
3.4.1.2	Base menu screen.....	39
3.4.1.3	Sub-menu.....	39
3.5	Program Structure.....	40
3.5.1	Sources files.....	40
3.5.2	Header Files.....	40
3.5.3	Database Tables.....	41
3.5.4	referenced environment variables.....	41
3.6	skymds00: Start-up and Login functions.....	41
3.6.1	Function: main.....	41
3.6.2	Function: initialise.....	41
3.6.3	Function: get_valid_login.....	41
3.7	skymds00: Main Menu.....	42
3.8	skymds00: Sub-menus.....	43
3.8.1	Menu Item “Paging” – skymds01.....	43
3.8.2	Menu Item “Client” – skymds00j.....	43
3.8.3	Sub-Menu “File” - Function: file_menu.....	43
3.8.4	Sub-Menu “Report” - Function: Report_menu.....	44
3.8.5	Sub-Menu “Customer” - Function: customer_menu.....	45
3.8.6	Sub-Menu “Utilities” - Function: utilities_menu.....	46
4	SKYMD01.....	47
4.1	skymds01 Overview.....	47
4.2	Running skymds01.....	47
4.2.1	Command line parameters.....	47
4.3	Program Description.....	47
4.4	Program Structure.....	47
4.4.1	Sources files.....	47
4.4.2	Header Files.....	49
4.4.3	Database Tables.....	50
4.4.4	skymds01 modules.....	50
4.4.5	referenced environment variables.....	51
4.5	Standard Functions.....	51
4.5.1	Function: main.....	51
4.5.2	Function: initialise.....	52
4.5.3	Function: general_logic.....	53
4.5.4	Function: base_logic.....	53
4.6	Diagram: skymds01: <System_Name> message input.....	55
4.7	Simple Paging Functions.....	56
4.7.1	Function: manual_logic.....	56
4.7.2	Function: answer_logic.....	57
4.7.2.1	<i>Function: answer_logic.....</i>	57
4.7.3	Function: contact_logic.....	58
4.7.4	Function: pager_selected_logic.....	61
4.7.4.1	<i>Function: pager_selected_logic: Contact processing.....</i>	61

4.7.4.2	<i>Function: pager_selected_logic - normal pager message processing.....</i>	62
4.7.5	<i>Function: get_pager_num.....</i>	63
4.7.6	<i>Function: get_pager_message.....</i>	64
4.7.7	<i>Function: general_pager_select.....</i>	64
4.7.7.1	<i>Master (Message entry) Key-press table.....</i>	65
4.8 Contact / Advanced processing functions.....		67
4.8.1	<i>Function: forms_logic.....</i>	67
4.8.1.1	<i>Database Tables used.....</i>	67
4.8.1.2	<i>Function: forms_logic.....</i>	67
4.8.1.3	<i>Function: FRM_forms_logic.....</i>	67
4.8.1.4	<i>Function: send_form.....</i>	67
4.8.2	<i>Function: search_logic.....</i>	68
4.8.2.1	<i>database tables used.....</i>	68
4.8.2.2	<i>Function: search_logic.....</i>	68
4.8.2.3	<i>Function: SCH_search_logic.....</i>	68
4.8.2.4	<i>Function: SCH_data_search.....</i>	68
4.8.3	<i>Function: call_patch_logic.....</i>	69
4.8.3.1	<i>Database tables used.....</i>	69
4.8.3.2	<i>Function: CPA_call_patch_logic.....</i>	69
4.8.3.3	<i>Function: CPA_process_call_patch.....</i>	69
4.8.4	<i>Function: advisory_logic.....</i>	70
4.8.4.1	<i>Database tables used.....</i>	70
4.8.4.2	<i>Function: advisory_logic.....</i>	70
4.8.4.3	<i>Function: ADV_advisory_logic.....</i>	70
4.8.5	<i>Function: external_database.....</i>	70
4.8.6	The four (remaining) external_database programs:	71
4.8.6.1	<i>Myer Grace Bros Facilities Response Centre (FRC).....</i>	71
4.8.6.2	<i>Safety <Company_Name> – Personal Safety monitoring systems.....</i>	72
4.8.6.3	<i><Company_1_Name> (STARTEL).....</i>	72
4.8.6.4	<i>V.I.P. – A home-services franchise.....</i>	73
4.8.7	<i>Function: MNT_maintenance_window.....</i>	73
4.8.8	Functions: MNT_xxx_maintenance / logic.....	74
4.8.8.1	<i>MNT_retrieval_logic – spawns skymds01a.....</i>	75
4.8.8.2	<i>MNT_away_maintenance – spawns skymds01e.....</i>	75
4.8.8.3	<i>MNT_booked_maintenance – spawns skymds01b.....</i>	75
4.8.8.4	<i>MNT_tmp_msg_maintenance – spawns skymds01l.....</i>	75
4.8.8.5	<i>MNT_talt_maintenance – spawns skymds01f.....</i>	75
4.8.8.6	<i>MNT_group_maintenance – spawns skymds01g.....</i>	75
4.8.8.7	<i>MNT_pager_details_maintenance – spawns skymds01p.....</i>	75
4.9 System Functions.....		76
4.9.1	<i>Function: event_processing – F4.....</i>	76
4.9.1.1	<i>Database tables used.....</i>	76
4.9.1.2	<i>Function: event_processing.....</i>	76
4.9.2	<i>Function: sys_ref_processing – Shift F4.....</i>	76
4.9.2.1	<i>Database table used.....</i>	77
4.9.2.2	<i>Function: sys_ref_processing.....</i>	77
4.9.3	<i>Function: reminder_retrieval – F5.....</i>	77
4.9.3.1	<i>database tables used.....</i>	77
4.9.3.2	<i>Function: REM_get_next_reminder.....</i>	77
4.9.3.3	<i>Function: reminder_screen.....</i>	78
4.10 Other Functions.....		78
4.10.1	<i>catch_sigusr1 (and catch_sigusr1_caller_drop).....</i>	78
4.10.2	<i>setup_signal_catching, catch_signal, catch_alarm, catch_term, catch.....</i>	78
4.10.3	<i>check_security_level.....</i>	78
4.10.4	<i>pop_help, push_help, set_help.....</i>	79
4.10.5	<i>display_infopack.....</i>	79
5 SKYMD01A TO SKYMD01P.....		80
5.1 program: skymds01a – Message Retrieval.....		80
5.1.1	<i>skymds01a: database table used.....</i>	80
5.1.2	<i>Function: (skymds01a) main.....</i>	80

5.1.3	Function: (skymds01a) initialise.....	81
5.1.4	Function: (skymds01a) msg_retrieval.....	81
5.2 Program: skymds01b – Booked Pager message Maintenance.....		81
5.2.1	Database tables used.....	82
5.2.2	Function: (skymds01b) main.....	82
5.2.3	Function: (skymds01b) initialise.....	82
5.2.4	Function: (skymds01b) booked Maintenance.....	82
5.2.4.1	<i>Menu Item “ADD”: calls booked_add.....</i>	83
5.2.4.2	<i>Menu Item “CHG”: Function booked_update.....</i>	84
5.2.4.3	<i>Menu Item “DEL”: Function booked_delete.....</i>	84
5.2.4.4	<i>Menu Item NEXT: Function booked_next</i>	
5.2.4.5	<i>Menu Item PREV: Function booked_prev.....</i>	84
5.3 Program skymds01d – Pager Holder Name Maintenance.....		84
5.3.1	Database tables used.....	84
5.3.2	Function: (skymds01d) main.....	84
5.3.3	Function: (skymds01d) holder_maint.....	84
5.3.4	Function: (skymds01d) holder_update.....	85
5.4 Program skymds01e – Away / Follow Me Maintenance.....		85
5.4.1	Database tables used.....	85
5.4.2	Function: (skymds01e) main.....	85
5.4.3	Function: (skymds01e) initialise.....	86
5.4.4	Function: (skymds01e) away_maint.....	86
5.5 Program skymds01f – Temporary Alternative Pager.....		87
5.5.1	Database tables used.....	87
5.5.2	Function: (skymds01f) main.....	87
5.5.3	Function: (skymds01f) initialise.....	87
5.5.4	Function: (skymds01f) alt_maint.....	87
5.6 Program: skymds01g – Group Paging Maintenance.....		88
5.6.1	Database tables used.....	88
5.6.2	Function: (skymds01g) main.....	88
5.6.3	Function: (skymds01g) initialise.....	88
5.6.4	Function: (skymds01g) pgroupt_maint.....	89
5.7 Program skymds01l – Temporary Message Maintenance.....		89
5.7.1	Database tables used.....	89
5.7.2	Function: (skymds01l) main.....	90
5.7.3	Function: (skymds01l) initialise.....	90
5.7.4	Function: (skymds01l) process_tmesg.....	90
5.7.5	Function: (skymds01l) add_tmsg and update_tmsg.....	91
5.7.6	Function: (skymds01l) delete_tmsg.....	91
5.8 Program: skymds01p – Pager Details Maintenance.....		91
5.8.1	Database tables used.....	91
5.8.2	Function: (skymds01p) main.....	91
5.8.3	Function: (skymds01p) initialise.....	92
5.8.4	Function: (skymds01p) display_pager_details.....	92
5.9 Diagram: skymds01a-p: maintenance function relationships.....		93
6 SKYMD08.....		94
6.1 skymds08 Overview.....		94
6.2 Running skymds08.....		94
6.2.1	Command line parameters.....	94
6.2.2	Example Command-line.....	94
6.3 Program Description.....		94
6.4 Program Structure.....		95
6.4.1	Sources files.....	95
6.4.2	Header Files.....	95
6.4.3	Database Tables.....	96
6.4.4	Environment Variables Referenced.....	96

6.5	skymds08: Major Functions.....	96
6.5.1	Function: main part 1: Initialise and get messages.....	96
6.5.2	Function: initialise.....	96
6.5.3	Function: get_valid_message(msg_length).....	97
6.5.4	Function: main part 2: Process the Received Message.....	97
6.6	log messages produced.....	98
6.6.1	mds_log Messages.....	98
6.6.2	Log Messages Written to stderr.....	98
6.6.3	Debug Mode Only: Log Messages Written to stderr.....	99
7	MDSEMGW.....	101
7.1	mdsemgw Overview.....	101
7.2	Running mdsemgw.....	101
7.2.1	Command line parameters.....	101
7.2.2	Example command-line.....	102
7.3	Program Description.....	102
7.4	Program Structure.....	103
7.4.1	Sources files.....	103
7.4.2	Header Files.....	103
7.4.3	Database Tables.....	103
7.4.4	referenced environment variables.....	103
7.5	mdsemgw: Major Functions.....	104
7.5.1	Function: main.....	104
7.5.2	Function: initialise.....	104
7.5.3	Function: check_block_email_table.....	104
7.5.4	check_shm_queue.....	104
7.5.5	Function: get_email_message.....	105
7.5.6	logging functions.....	105
7.6	log messages produced.....	106
7.6.1	While in debug mode only.....	106
7.6.2	Always displayed.....	107
7.7	Stopping mdsemgw.....	107
8	MDSEM01.....	108
8.1	mdsem01 Overview.....	108
8.2	Running mdsem01.....	108
8.2.1	Command line parameters.....	108
8.2.2	Example command-line.....	108
8.3	Program Description.....	108
8.4	Program Structure.....	109
8.4.1	Sources files.....	109
8.4.2	Header Files (Non-Database).....	109
8.4.3	Database Header files.....	109
8.4.4	Necessary (referenced) environment variables.....	110
8.5	mdsem01: Major Functions.....	110
8.5.1	Function: main.....	110
8.5.2	Function: initialise.....	110
8.5.3	Function: process_message.....	111
8.6	log messages produced.....	112
8.6.1	MDS_LOG messages.....	112
8.6.2	Log messages written to stderr.....	112
8.6.3	debug mode only: Log messages written to stderr.....	113
8.7	Stopping mdsem01.....	113
9	MDSEM02.....	114
9.1	mdsem02 Overview.....	114

9.2	Running mdsem02.....	114
9.2.1	Command line parameters.....	114
9.2.2	Example command-line.....	114
9.3	Program Description.....	114
9.4	Program Structure.....	115
9.4.1	Sources files.....	115
9.4.2	Non- Database Header Files.....	115
9.4.3	Database Table.....	115
9.4.4	referenced environment variables.....	116
9.5	mdsem02: Major Functions.....	116
9.5.1	Function: main.....	116
9.5.2	Function: Initialise.....	116
9.5.3	Function: process_message.....	116
9.5.4	Other Functions.....	118
9.5.4.1	<i>get_email_files.....</i>	118
9.5.4.2	<i>MIME_get_basicpart.....</i>	118
9.5.4.3	<i>Send_Messages.....</i>	118
9.6	log messages produced.....	118
9.6.1	mds_log log messages.....	118
9.6.2	MDS_LOG_DEBUG Log messages.....	119
9.6.3	stderr – debug mode only. Log messages.....	119
9.7	Stopping mdsem02.....	120
9.8	Appendix to mdsem02 – MIME Record structure.....	121
10	MDSEM03.....	122
10.1	mdsem03 Overview.....	122
10.2	Running mdsem03.....	122
10.2.1	Command line parameters.....	122
10.2.2	Example command-line.....	122
10.3	Program Description.....	123
10.3.1	Logic of a BulkSMS instruction file.....	123
10.3.1.1	<i>A Bulk SMS instruction file example.....</i>	123
10.3.1.2	<i>Bulk-SMS instruction file tags.....</i>	124
10.4	Program Structure.....	124
10.4.1	Sources files.....	124
10.4.2	Non- Database Header Files.....	125
10.4.3	Database Table.....	125
10.4.4	referenced environment variables.....	125
10.5	mdsem03: Major Functions.....	126
10.5.1	Function: main.....	126
10.5.2	Function: Initialise.....	126
10.5.3	Function: process_message.....	126
10.5.3.1	<i>Similar to mdsem02, the single-SMS daemon.....</i>	126
10.5.3.2	<i>Specialised Bulk SMS processing.....</i>	127
10.5.4	Function: process_bulksms_file.....	128
10.5.4.1	<i>Processing the <#BULKMSG#> tag: The SMS message Text.....</i>	128
10.5.4.2	<i>Processing the <#BULKID#> tag: the destination mobile-phone numbers.....</i>	128
10.5.4.3	<i>Processing the <#PRIORITY#> tag.....</i>	128
10.5.4.4	<i>Processing the <#UNIQUEID#> tag.....</i>	128
10.5.5	Function: check_file_content.....	129
10.5.6	Other Functions.....	129
10.5.6.1	<i>process_bulksms_mesg.....</i>	129
10.5.6.2	<i>check_unique_msg.....</i>	129
10.5.6.3	<i>extractBatchNoAndPin.....</i>	129
10.6	Stopping mdsem03.....	129
11	MDSMS01.....	130

11.1	mdssms01 Overview.....	130
11.2	Running mdssms01.....	130
11.2.1	Command line parameters.....	130
11.2.2	Example command-line.....	130
11.3	Program Description.....	130
11.4	Program Structure.....	131
11.4.1	Sources files.....	131
11.4.2	Header Files.....	131
11.4.3	Database Table.....	132
11.4.4	referenced environment variables.....	132
11.5	mdssms01: Major Functions.....	132
11.5.1	Function: main.....	132
11.5.2	Function: Initialise.....	132
11.5.3	Function: process_pqu_recs.....	133
11.5.4	Function: read_pqu_rec.....	133
11.5.5	Function: queue_write_rec.....	133
11.5.6	Function: update_pqu_header.....	133
11.6	Stopping mdssms01.....	133
12	SKYMDS96A.....	134
12.1	skymds96a Overview.....	134
12.2	Running skymds96a.....	134
12.2.1	Command line parameters.....	134
12.2.2	Example command-line.....	134
12.3	Program Description.....	134
12.4	Program Structure.....	135
12.4.1	Sources files.....	135
12.4.2	Header Files.....	135
12.4.3	Database Tables.....	136
12.4.4	referenced environment variables.....	136
12.5	skymds96a: Major Functions.....	136
12.5.1	Function: main.....	136
12.5.2	Function: Initialise.....	136
12.5.3	Function: process_requests.....	137
12.5.4	Function: REQ_id_page.....	138
12.5.5	check_pager_and_security.....	138
12.5.6	Other functions.....	138
12.5.6.1	get_next_select_mesg.....	138
12.5.6.2	bulkq_db_update.....	138
12.5.6.3	check_security.....	138
12.5.6.4	validate_message.....	138
12.5.6.5	ss_server_served.....	138
12.5.6.6	setup_queue_rec.....	138
12.5.6.7	validate_user.....	139
12.5.7	Functions No longer Used.....	139
12.6	log messages produced.....	139
12.7	Stopping skymds96a.....	139
13	SKYMDS02.....	140
13.1	skymds02 Overview.....	140
13.2	Running skymds02.....	140
13.2.1	Command Line Parameters.....	140
13.3	Program Description.....	140
13.4	Program Structure.....	141
13.4.1	Source Files.....	141
13.4.2	Header Files.....	141

13.4.3	Database Tables.....	142
13.4.4	Environment Variables Referenced.....	142
13.5 skymds02: Major Functions.....		142
13.5.1	Function: main.....	142
13.5.2	Function: initialise.....	143
13.5.3	Function: await_login.....	143
13.5.4	Function: get_valid_message.....	144
13.5.5	queue_main.....	144
13.5.6	Function: stat_opers.....	145
13.6 Log Messages Produced.....		145
13.6.1	debug-mode only: Log Messages written to stderr.....	145
13.6.2	Log messages written to .../logs/mdspet???.txt and stderr.....	148
13.6.3	Log messages written to mdslog.log (and usually stderr).....	149
13.7 Terminating skymds02.....		150
13.7.1	Send the sig_term signal.....	150
13.7.2	kill -9 PID.....	150
13.7.3	Log-on with an “EXIT” pager number.....	150
13.7.4	Other.....	150
14 SKYMDS70.....		151
14.1 skymds70 Overview.....		151
14.2 Running skymds70.....		151
14.2.1	command line parameters.....	151
14.2.2	example command line.....	151
14.2.3	Note re: IXON and IOFF.....	151
14.3 Program Description.....		152
14.4 Program structure.....		152
14.4.1	Source files.....	152
14.4.2	Header files (non-database).....	152
14.4.3	Database tables.....	153
14.4.4	Environmental variables referenced.....	153
14.5 skymds70: Major Functions.....		153
14.5.1	Function: main().....	153
14.5.2	Function: Initialise.....	154
14.5.3	Function: get_valid_message.....	155
14.5.4	Function: validate_pager_message.....	155
14.5.5	Function: validate_cap_message.....	155
14.5.6	Function: queue_main.....	155
14.5.7	Function: are_you_alive.....	156
14.5.8	Function: stat_opers.....	156
14.5.9	Other Functions.....	156
14.5.9.1	validate_user.....	156
14.5.9.2	check_pager_and_security.....	156
14.6 log messages produced.....		156
14.7 Stopping skymds70.....		156
15 SKYMDS18.....		157
15.1 skymds18 Overview.....		157
15.2 Running skymds18.....		157
15.2.1	Command line parameters.....	157
15.2.2	Example command-line.....	157
15.3 Program Description.....		157
15.4 Program Structure.....		157
15.4.1	Sources files.....	158
15.4.2	Header Files.....	158

15.4.3	Database Tables.....	159
15.4.4	referenced environment variables.....	159
15.5 skymds18: Major Functions.....		159
15.5.1	Function: main.....	159
15.5.2	Function: initialise.....	160
15.5.3	Function: get_valid_message.....	161
15.5.4	Functions: The 28 Wang message processing functions (in brief).....	161
15.5.4.1	<i>proc_test_page_num(mess_len)</i>	162
15.5.4.2	<i>proc_test_page_cap(mess_len)</i>	162
15.5.4.3	<i>proc_pager_add()</i>	162
15.5.4.4	<i>proc_pager_change()</i>	162
15.5.4.5	<i>proc_pager_delete()</i>	163
15.5.4.6	<i>proc_line_add()</i>	163
15.5.4.7	<i>proc_line_change()</i>	163
15.5.4.8	<i>proc_line_delete()</i>	163
15.5.4.9	<i>proc_pager_group_add()</i>	164
15.5.4.10	<i>proc_pager_group_change()</i>	164
15.5.4.11	<i>proc_pager_group_delete()</i>	164
15.5.4.12	<i>proc_client_add()</i>	164
15.5.4.13	<i>proc_client_change()</i>	164
15.5.4.14	<i>proc_client_delete()</i>	165
15.5.4.15	<i>proc_pin_add()</i>	165
15.5.4.16	<i>proc_pin_change()</i>	165
15.5.4.17	<i>proc_pin_delete()</i>	165
15.5.4.18	<i>proc_email_add()</i>	165
15.5.4.19	<i>proc_email_change()</i>	166
15.5.4.20	<i>proc_email_delete()</i>	166
15.5.4.21	Note on the 'Delay Pager' Facility.....	166
15.5.4.22	<i>proc_delay_pager_add()</i>	166
15.5.4.23	<i>proc_delay_pager_change()</i>	166
15.5.4.24	<i>proc_delay_pager_delete()</i>	167
15.5.4.25	<i>proc_aparty_add()</i>	167
15.5.4.26	<i>proc_aparty_change()</i>	167
15.5.4.27	<i>proc_aparty_delete()</i>	167
15.6 log messages produced.....		167
15.7 Stopping skymds18.....		167

16 THE HISTORY SERVER, HISTORY DATA.....169

16.1 Overview.....		169
16.2 History.....		169
16.2.1	Machine – History Server.....	169
16.2.2	History Data.....	169
16.3 History concepts.....		170
16.4 Brief review of History File formats.....		170
16.4.1	Primary History records.....	170
16.4.2	Creation and Handling the History Files.....	170
16.4.2.1	<i>Creating History Files</i>	170
16.4.2.2	<i>Storing History Files</i>	171
16.4.2.3	<i>Merging History Files</i>	171
16.4.3	Filling, population of the history data files.....	171
16.4.4	Primary History File (e.g. hi20020304.dat).....	171
16.4.5	The Secondary History File (e.g. hi20020304.txt).....	172
16.5 A Primary and Secondary History file example.....		174
16.6 Detailed Description of The History Record example.....		175
16.6.1.1	<i>The Initial Message's History</i>	175
16.6.1.2	<i>'Spawned' Pager and Roster message history</i>	175
16.6.1.3	<i>Detailed Roster Message History</i>	176
16.6.1.4	<i>Email contact generated by the Roster and 'previous' email</i>	176
16.6.1.5	<i>What is happening here?</i>	176

16.6.1.6	<i>Another Email and SMS triggered by the Group's pager message.....</i>	177
16.6.1.7	<i>SMS 'return-receipt', detailing which carrier used.....</i>	177
hicccyymmd – Primary History database tables.....		178
Secondary History Data Layout.....		179
16.7 Diagram: History Generating systems in <System_Name>.....		181
17 CDR - CALL DETAIL RECORD.....		183
17.1.1	CDR File Structure.....	183
17.1.2	CDR File Structure Matrix.....	185
18 SKYMD500J - CLIENT MAINTENANCE.....		188
18.1 program & screen initialisation.....		188
18.1.1	screen construction.....	188
18.2 Function: file_edit.....		189
18.2.1.1	<i>General usage keys and functions.....</i>	189
18.2.1.2	<i>Contact Management Functions.....</i>	189
18.2.1.3	<i>Hidden, un-noted, functions.....</i>	189
18.3 Function: select_contact.....		190
18.4 General form of Contact Maintenance Functions.....		190
18.5 adv_advisory_details: Advisory Maintenance.....		191
18.5.1	Program & Screen initialisation.....	191
18.5.2	Advisory Maintenance: Function: adv_process_advisory.....	191
18.6 cpa_call_patch_details.....		192
18.6.1	Screen Initialisation.....	192
18.6.2	Call - Patch Maintenance: Function: cpa_process_call_patch.....	192
18.7 eml_email_details.....		193
18.7.1	Screen Initialisation.....	193
18.7.2	Email Maintenance: Function: eml_process_email.....	193
18.8 esc_escalation_details.....		194
18.8.1	Program & Screen initialisation.....	194
18.8.2	Escalation Maintenance: Function: esc_process_Escalation.....	194
18.9 fax_fax_details.....		195
18.9.1	Program & Screen initialisation.....	195
18.9.2	Fax Maintenance: Function: fax_process_fax.....	195
18.10 frm_form_details.....		196
18.10.1	Program & Screen initialisation.....	196
18.10.2	Form Maintenance: Function: frm_process_Form.....	196
18.11 grp_group_details.....		197
18.11.1	Program & Screen initialisation.....	197
18.11.2	Group Maintenance: Function: grp_process_group.....	197
18.12 rem_reminder_details.....		198
18.12.1	Program & Screen initialisation.....	198
18.12.2	Reminder Maintenance: Function: rem_process_rem.....	198
18.13 ros_roster_details.....		199
18.13.1	Program & Screen initialisation.....	199
18.13.2	Roster Maintenance: Function: ros_process_Roster.....	199
18.14 sch_search_par_details.....		200
18.14.1	Program & Screen initialisation.....	200
18.14.2	Search data Maintenance: Function: schp_process_schp.....	200
18.15 sch_search_dat_details.....		201
18.15.1	Program & Screen initialisation.....	201
18.15.2	Search data Maintenance: Function: schd_process_schd.....	201
18.16 tms_tms_details.....		202
18.16.1	Program & Screen initialisation.....	202
18.16.2	Text Message Maintenance: Function: tms_process_tms.....	202

18.17 text_txt_details.....	203
18.17.1 Program & Screen initialisation.....	203
18.17.2 'Text' message Maintenance: Function: txt_process_text.....	203
19 FILE MENU PROGRAMS.....	204
19.1 "Username / password Maintenance": skymds00b.....	205
19.1.1 Program & Screen initialisation.....	205
19.1.2 Function: process_screen.....	205
19.2 "Operator Bulletin maintenance": skymds00d.....	206
19.2.1 Program & Screen initialisation.....	206
19.2.2 Function: process_screen.....	206
19.3 "public holiday maintenance": skymds00p.....	207
19.3.1 Program & Screen initialisation.....	207
19.3.2 Function: process_screen.....	207
19.4 "Function Code Maintenance": skymds39.....	208
19.4.1 Program & Screen initialisation.....	208
19.4.2 Function: process_screen.....	208
19.5 "Security Group Maintenance": skymds32.....	209
19.5.1 Program & Screen initialisation.....	209
19.5.2 Function: process_screen.....	209
19.6 "Audit Trail": skymds77.....	210
19.6.1 Program & Screen initialisation.....	210
19.6.2 Function: process_screen.....	210
"Reminder Parameter Maintenance": skymds34.....	211
19.6.3 Program & Screen initialisation.....	211
19.6.4 Function: process_screen.....	211
19.7 "Email Blocking Maintenance": mdsemlblk.....	212
19.7.1 Program & Screen initialisation.....	212
19.7.2 Function: select_report.....	212
19.8 "Destination File Maintenance": mdsdescri.....	213
19.8.1 Program & Screen initialisation.....	213
19.8.2 Function: run_maint_scr.....	213
20 REPORT MENU.....	214
20.1 "Set Printer".....	215
20.2 "Initialise Printer".....	215
20.3 Report Generator (History Data): skymds46.....	216
20.3.1 Program & Screen initialisation.....	216
20.3.2 Function: process_report.....	216
20.3.3 Function: spawn_report.....	216
20.4 Report Generator: sub programs.....	217
20.4.1 skymds46a: Event & System Reference Number Reports.....	217
20.4.2 skymds46b: Contact Message Report.....	217
20.4.3 skymds46c: Pager Message Report.....	217
20.4.4 skymds46d: Line Message Report.....	217
20.5 Pager Stats Report: skymds40.....	218
20.5.1 Program & Screen initialisation.....	218
20.5.2 Functions: process_report, generate_report.....	218
20.5.3 Functions: process_client, process_pager, print_pager.....	218
20.6 Detailed Line Report: skymds47 (and skymds47a).....	219
20.6.1 Program & Screen initialisation.....	219
20.6.2 Function: select_report.....	219
20.6.3 Script: dlr.script (called by email_report()).....	219
20.6.4 Program: skymds47a.....	219
20.7 Answer-Phrase report: skymds48 (and skymds48a).....	220
20.7.1 Program & Screen initialisation.....	220

20.7.2	Function: select_report and spawn_report.....	220
20.7.3	Program: skymds48a.....	220
20.8 “Batch Fax Setup”: skymds19.....		221
20.8.1	Program & Screen initialisation.....	221
20.8.2	Function: process_screen.....	221
20.9 “Batch Fax List”: skymds46e.....		222
20.9.1	Program & Screen initialisation.....	222
20.9.2	Function: print_headings.....	222
20.9.3	Function: process_batch_fax.....	222
20.10 “Batch Email Setup”: skymds19a.....		223
20.10.1	Program & Screen initialisation.....	223
20.10.2	Function: process_screen.....	223
21 CUSTOMER MENU.....		224
21.1 Customer Menu: Sub-Menus.....		224
21.1.1	sub-menu functionality.....	224
21.1.2	“V.I.P.” Menu.....	225
21.1.3	“<Company_1_Name>/X” Menu.....	225
21.1.4	“F.R.C.” Menu <Company_3_Name>.....	225
21.1.5	“Safety <Company_Name>” Menu.....	225
21.2 Customer Menu: Trade/Service Code Maint: skymds30.....		226
21.2.1	Program & Screen initialisation.....	226
21.2.2	Function: process_screen.....	226
22 UTILITIES MENU.....		227
22.1 Watch Queue (60 and 5 Seconds): mdswq.....		227
22.1.1	Program & Screen initialisation.....	227
22.1.2	Functions: main and process_dests.....	227
22.2 “Operator List Report”: skymds33.....		228
22.2.1	Program initialisation.....	228
22.2.1.1	In skymds00’s utility_menu() function.....	228
22.2.1.2	skymds33’s initialise() function.....	228
22.2.2	Report Generation: Function process_print.....	228
22.2.3	Function: print_oper.....	228
22.3 “Terminal Monitoring”: mdstymon.....		229
22.3.1	Program & Screen initialisation.....	229
22.3.2	Monitor the Operator: Functions: ttymon_select and turn_echo_on.....	229
22.3.2.1	Function: ttymon_select.....	229
22.3.2.2	turn_echo_on.....	229
22.4 “Network Delay Monitoring”: skymds83c.....		230
22.4.1	Program & Screen initialisation.....	230
22.4.2	Function: process_screen.....	230
22.5 “Network Connection Maintenance”: skymds79.....		231
22.5.1	Program & Screen initialisation.....	231
22.5.2	Function: process_screen.....	231
22.6 “Mobile Phone Maintenance”: skymds65.....		232
22.6.1	Program & Screen initialisation.....	232
22.6.2	Function: process_screen.....	232
22.7 “Port Maintenance”: skymds84.....		233
22.7.1	Program & Screen initialisation.....	233
22.7.2	Function: process_screen.....	233
22.8 “Alarm Maintenance”: skymds92.....		234
22.8.1	Program & Screen initialisation.....	234
22.8.2	Function: process_screen.....	234
22.9 “Customer Program Maintenance”: skymds88.....		235
22.9.1	Program & Screen initialisation.....	235
22.9.2	Function: process_screen.....	235

22.10 “Fax Delivery Status”: skymds78.....	236
22.10.1 Program & Screen initialisation.....	236
22.10.2 Function: process_screen.....	236
23 SKYMD20.....	237
23.1 skymds20 Overview.....	237
23.2 Running skymds20.....	237
23.2.1 Command line parameters.....	237
23.2.2 Example command-line.....	237
23.3 Program Description.....	237
23.4 Program Structure.....	238
23.4.1 Sources files.....	238
23.4.2 Header Files.....	238
23.4.3 Database Tables.....	239
23.4.4 referenced environment variables.....	239
23.5 Important data types.....	240
23.6 skymds20: Major Functions.....	240
23.6.1 Function: main.....	240
23.6.2 Function: initialise.....	240
23.6.3 Function: get_next_message.....	241
23.6.4 Function: QU_queue_read_mesg.....	241
23.6.5 Function: update_queue , QU_queue_update_hdr.....	241
23.6.6 Function: queue_all.....	241
23.6.7 Function: queue_all_pagers.....	242
23.6.8 Function: queue_single_pager.....	242
23.6.9 Function: queue_single_dest.....	243
23.6.10 Function: QU_queue_write_mesg.....	243
23.7 log messages produced.....	243
23.8 Stopping skymds20.....	243
24 MDSALARM.....	244
24.1 mdsalarm Overview.....	244
24.2 Running mdsalarm.....	244
24.2.1 Command line parameters.....	244
24.2.2 Example command-line.....	244
24.3 Program Description.....	245
24.4 Program Structure.....	245
24.4.1 Sources files.....	245
24.4.2 Header Files.....	245
24.4.3 Database Tables.....	246
24.4.4 referenced environment variables.....	246
24.5 mdsalarm: Major Functions.....	247
24.5.1 Function: main.....	247
24.5.2 Function: initialise.....	247
24.5.3 Function: ALARM_process_alarm.....	248
24.5.4 Function: ALARM_send_message_local.....	248
24.5.5 Function: ALARM_send_message_remote.....	248
24.5.6 Function: check_processes.....	248
24.5.7 Function: check_queues.....	249
24.5.8 Function: check_eml_queues.....	249
24.5.9 Function: check_acdr_rec.....	249
24.5.10 Function: check_queue_mq.....	249
24.5.11 Function: check_ss_bulkq_db.....	250
24.5.12 Function: check_activity.....	250
24.5.13 Function: check_disk.....	250
24.5.14 Function: check_disable_alarm.....	250
24.5.15 Function: check_grncdr_record.....	250

24.5.16	Function: check_hist_files.....	251
24.5.17	Function: check_network_connection_file.....	251
24.5.18	Function: check_reminder.....	251
24.5.19	Function: check_network_delay_file.....	251
24.6	log messages produced.....	252
24.7	Stopping mdsalarm.....	252
25	SKYMD23.....	253
25.1	skymds23 Overview.....	253
25.2	Running skymds23.....	253
25.2.1	Command line parameters.....	253
25.2.2	Example command-line.....	253
25.3	Program Description.....	254
25.4	Program Structure.....	254
25.4.1	Sources files.....	254
25.4.2	Header Files.....	254
25.4.3	Database Tables.....	254
25.4.4	referenced environment variables.....	254
25.5	skymds23: Major Functions.....	255
25.5.1	Function: main.....	255
25.5.2	Function: initialise.....	255
25.5.3	Function: get_next_mess.....	255
25.5.4	Function: mess_to_lan.....	256
25.5.5	Function: mnt_queue_open_for_read.....	256
25.5.6	Function: update_queue.....	256
25.5.7	Function: connect_to_lan.....	257
25.6	log messages produced.....	257
25.7	Stopping skymds23.....	257
26	SKYMD24.....	258
26.1	skymds24 Overview.....	258
26.2	Running skymds24.....	258
26.2.1	Command line parameters.....	258
26.2.2	Example command-line.....	258
26.3	Program Description.....	258
26.3.1.1	<i>Getting the updates.....</i>	258
26.3.1.2	<i>Implementing updates.....</i>	259
26.4	Program Structure.....	259
26.4.1	Source files.....	259
26.4.2	Header Files.....	259
26.4.3	Database Tables.....	261
26.4.4	referenced environment variables.....	262
26.5	skymds24: Major Functions.....	262
26.5.1	Function: main.....	262
26.5.2	Function: initialise.....	262
26.5.3	Function: connect_to_lan.....	262
26.5.4	Function: process_inter_machine_update.....	263
26.5.5	Function: check_for_work.....	263
26.5.6	Function: get_next_mess.....	263
26.5.7	Function: validate_transaction.....	263
26.5.8	Function: mnt_queue_re_write.....	263
26.5.9	Function: process_transaction.....	264
26.5.10	Functions: skymds24 Handler Functions.....	265
26.5.10.1	<i>Function: process_clsts_rec.....</i>	266
26.5.10.2	<i>Function: process_destm_rec.....</i>	266
26.5.10.3	<i>Function: process_dests_rec.....</i>	266
26.5.10.4	<i>Function: process_facil_rec.....</i>	266

26.5.10.5	Function: process_g1_rec.....	267
26.5.10.6	Function: process_hist_rec.....	267
26.5.10.7	Function: process_hsec_rec.....	267
26.5.10.8	Function: process_line_client_rec.....	268
26.5.10.9	Function: process_pager_rec.....	268
26.5.10.10	Function: process_pasts_rec.....	268
26.5.10.11	Function: process_pubh_rec.....	269
26.5.10.12	Function: process_rec.....	269
26.5.10.13	Function: process_rem_rec.....	269
26.5.10.14	Function: process_remp_rec.....	270
26.5.10.15	Function: process_rename_file.....	270
26.5.10.16	Function: process_schd_rec.....	270
26.5.10.17	Function: process_secur_rec.....	270
26.5.10.18	Function: process_text_file.....	271
26.5.10.19	Function: process_timev_rec.....	271
26.5.10.20	Functions: add_generic, update_generic, change_generic, and delete_generic.....	271
26.5.11	Function: display_mntq_rec.....	271
26.6	log messages produced.....	272
26.7	Stopping skymds24.....	272
27	SKYMD28.....	273
27.1	skymds28 Overview.....	273
27.2	Running skymds28.....	273
27.2.1	Command line parameters.....	273
27.2.2	Example command-line.....	273
27.3	Program Description.....	273
27.4	Program Structure.....	274
27.4.1	Sources files.....	274
27.4.2	Header Files.....	274
27.4.3	Database Tables.....	275
27.4.4	referenced environment variables.....	275
27.5	skymds28: Major Functions.....	275
27.5.1	Function: main.....	275
27.5.2	function: initialise.....	276
27.5.3	function: process_connect.....	276
27.6	Top level History Request Handlers.....	276
27.6.1	Function: process_prim_read.....	276
27.6.2	Function: process_prim_write.....	277
27.6.3	Function: process_hs_read_next.....	277
27.6.4	Function: process_hs_write.....	277
27.6.5	Function: process_write_retrieval.....	277
27.6.6	Function: process_set_mode.....	278
27.7	Lower Level ‘utility’ history functions.....	278
27.7.1	Function: HIST_start_session.....	278
27.7.2	Function: check_hist_and_sec.....	279
27.7.3	Function: check_hist_and_sec.....	279
27.7.4	Function: get_next_mess.....	279
27.7.5	Function: HIST_read_prim [primary History].....	279
27.7.6	9 Functions: Specialised Primary Hist. Read functions [primary History].....	280
27.7.7	Function: HIST_write_prim [primary History].....	280
27.7.8	Function: HIST_posn_sec [Secondary History].....	280
27.7.9	Function: HIST_read_sec_next [Secondary History].....	281
27.7.10	Function: HIST_write_sec [Secondary History].....	281
27.7.11	Function: HIST_write_retrv_to_hsec [Secondary History].....	281
27.8	log messages produced.....	282
27.9	Stopping skymds28.....	282
28	SKYMD03.....	284

28.1	skymds03 Overview.....	284
28.2	Running skymds03.....	284
28.2.1	Command-Line Parameters.....	284
28.2.2	Example Command-line.....	285
28.3	Program Description.....	285
28.4	Program structure.....	286
28.4.1	Source files.....	286
28.4.2	header files.....	287
28.4.3	Database tables.....	287
28.5	skymds01: Major Functions.....	288
28.5.1	Function: main.....	288
28.5.2	function initialise.....	289
28.5.3	Function: init_network_connection.....	289
29	SKYMD03 SUB-MODULES.....	291
29.1.1	Overview.....	291
29.2	Module: mdssende.c: Send Emails.....	291
29.2.1	Function: init_email.....	291
29.2.2	Function: finalise_email.....	291
29.2.3	Function: mess_to_email.....	292
29.3	Module: mdssendf.c: Send Faxes.....	292
29.3.1	Function: init_fax.....	292
29.3.2	Function: mess_to_fax.....	292
29.4	Module: mdssendp.c: Send PET messages.....	293
29.4.1	Function: init_pet.....	293
29.4.2	Function: finalise_pet.....	293
29.4.3	Function: mess_to_pet.....	293
29.4.4	Function: send_message.....	294
29.4.5	Function: logon_pet.....	294
29.4.6	Function: disconnect_modem.....	294
29.5	Module mdssendm.c: Send MERP messages.....	295
29.5.1	Function: init_merp.....	295
29.5.2	Function: mess_to_merp.....	295
29.5.3	Function: reset_sequ_nbr.....	295
29.6	Module mdssenda.c: Send MTP messages.....	296
29.6.1	Function: init_mtp_serv.....	296
29.6.2	Function: mess_to_mtp_serv.....	296
29.7	Module mdssendw.c: Messages to an XACOM encoder.....	297
29.7.1	Function: init_equatorial.....	297
29.7.2	Function: mess_to_equatorial.....	297
29.8	Module mdssendx.c: Messages to XACOM encoders (with reply).....	298
29.8.1	Function: init_xacom.....	298
29.8.2	Function: mess_to_xacom.....	298
29.9	Module mdssendl.c: Send Messages to other <System_Name> sites.....	298
29.9.1	Function: init_remote_03.....	298
29.9.2	Function: init_local_03.....	299
29.9.3	Function: mess_to_remote_03.....	299
29.10	Module mdssendv.c: Add Reminders.....	299
29.10.1	Function: init_reminders.....	299
29.10.2	Function: mess_to_reminders.....	299
29.11	Module mdssendz.c: Sending SMS Messages.....	299
29.11.1	Function: mess_to_smpp_server.....	300
29.11.2	Function: update_msg_id_and_deliv_status.....	300
29.11.3	Function: check_mob_num.....	300
29.12	Module mdssendn.c: Messages to Telecom New Zealand.....	301
29.12.1	Function: init_pacnet.....	301
29.12.2	Function: Mess_to_pacnet.....	301

29.13 Module mdssendd.c – Send Test / dummy messages.....	301
30 SKYMDS25.....	302
30.1 skymds25 Overview.....	302
30.2 Running skymds25.....	302
30.2.1 Command line parameters.....	302
30.2.2 Example command-line.....	302
30.3 Program Description.....	303
30.4 Program Structure.....	303
30.4.1 Sources files.....	303
30.4.2 Header Files.....	303
30.4.3 Database Tables.....	304
30.4.4 referenced environment variables.....	304
30.5 skymds25: Major Functions.....	304
30.5.1 Function: main.....	304
30.5.2 Function: initialise.....	305
30.5.3 Function: get_next_mess.....	305
30.5.4 Function: process_transaction.....	305
30.5.5 Function: process_tmp_msg_trans.....	306
30.5.6 Function: process_sms_phone_trans.....	306
30.5.7 Function: process_holder_name_trans.....	307
30.5.8 Function: process_answer_phrase_trans.....	307
30.5.9 Function: process_gl3000_error.....	308
30.5.10 Function: follow_me_trans.....	308
30.5.11 Function: send_transaction and send_trans_rec.....	308
30.5.11.1 Function: send_trans_rec.....	308
30.5.12 Function: update_queue.....	309
30.6 log messages produced.....	309
30.7 Stopping skymds25.....	309
31 SKYMDS10.....	310
31.1 skymds10 Overview.....	310
31.2 Running skymds10.....	310
31.2.1 Command line parameters.....	310
31.2.2 Example command-line.....	310
31.3 Program Description.....	311
31.4 Program Structure.....	311
31.4.1 Sources files.....	311
31.4.2 Header Files.....	312
31.4.3 Database Tables.....	312
31.4.4 referenced environment variables.....	312
31.5 skymds10: Major Functions.....	313
31.5.1 Function: main.....	313
31.5.2 function: initialise.....	313
31.5.3 Function: tnpp_main.....	313
31.6 The Five state-dependent TNPP processing functions.....	314
31.6.1 Function: tnpp_init_state.....	314
31.6.2 Function: tnpp_await_ENQ_state.....	314
31.6.3 Function: tnpp_ready_state.....	314
31.6.4 Function: tnpp_transmit_state.....	315
31.6.5 Function: tnpp_await_tx_response_state.....	315
31.7 Support Functions.....	316
31.7.1 Function: tnpp_setup.....	316
31.7.2 Function: tnpp_send_host.....	316
31.7.3 Function: tnpp_get_response.....	316
31.7.4 Function: dispatch_tnpp_packet.....	318
31.7.5 Function: check_mess_from_ip.....	318

31.7.6	Function: get_mess_from_ip.....	318
31.7.7	Function: tnpp_transmit_packet.....	319
31.7.8	Function: tnpp_transmit_null.....	319
31.7.9	Function: tnpp_status.....	319
31.7.9.1	<i>On receipt of a packet from the external TNPP site.....</i>	319
31.7.9.2	<i>Other responses from the external TNPP site.....</i>	320
31.7.9.3	<i>'Don't respond' status conditions.....</i>	320
31.7.10	Function: process_incomming_packet.....	320
31.8	Log messages produced.....	321
31.9	Stopping skymds10.....	321
32	SKYMDS11T (THE FORMER SKYMDS11A).....	322
32.1	skymds11t Overview.....	322
32.2	Running skymds11t.....	322
32.2.1	Command line parameters.....	322
32.2.2	Example command-line.....	322
32.3	Program Description.....	323
32.4	Program Structure.....	323
32.4.1	Sources files.....	323
32.4.2	Header Files.....	323
32.4.3	Database Tables.....	323
32.4.4	referenced environment variables.....	324
32.5	skymds11t: Major Functions.....	324
32.5.1	Function: main.....	324
32.5.2	Function: initialise.....	324
32.5.3	Function: process_smpp.....	325
32.5.4	Function: connect_and_bind_to_smsc.....	326
32.5.5	Function: connect_to_lan.....	326
32.5.6	Function: wait_either_input.....	326
32.5.7	Function: check_enquire_<Company_Name>.....	326
32.5.8	Function: get_next_mess.....	326
32.5.9	Function: mess_to_smsc.....	327
32.5.10	Function: reply_to_host.....	327
32.6	log messages produced.....	327
32.7	Stopping skymds11t.....	327
33	SKYMDS27.....	328
33.1	skymds27 Overview.....	328
33.2	Running skymds27.....	328
33.2.1	Command line parameters.....	328
33.2.2	Example command-line.....	328
33.3	Program Description.....	329
33.4	Program Structure.....	329
33.4.1	Sources files.....	329
33.4.2	Header Files.....	330
33.4.3	Database Tables.....	331
33.4.4	referenced environment variables.....	332
33.5	skymds27: Major Functions.....	332
33.5.1	Function: main.....	332
33.5.2	Function: initialise.....	332
33.5.3	Function: process_contact.....	333
33.5.4	Functions: process_ros and roster_logic.....	333
33.5.4.1	<i>Function: roster_logic.....</i>	333
33.5.5	Functions: process_esc and escalation_logic.....	334
33.5.5.1	<i>Function: escalation_logic.....</i>	334
33.5.6	Function: pager_logic.....	334
33.5.7	Function: process_grp and group_logic.....	335

33.5.7.1	<i>Function: group_logic</i>	335
33.5.8	Function: fax_logic.....	335
33.5.9	Functions: process_frm and forms_logic.....	335
33.5.10	Function: tms_logic.....	335
33.5.11	Functions: txt_logic and process_text.....	336
33.5.11.1	<i>Function: process_text</i>	336
33.5.12	Function: reminder_logic.....	336
33.5.12.1	<i>Function: generate_reminder</i>	336
33.5.12.2	<i>Function: REM_addReminder</i>	336
33.5.13	Function: phone_logic.....	337
33.5.13.1	<i>phn_generate_reminder</i>	337
33.5.14	Function: email_logic.....	337
33.5.15	Function: supervisor_reminder.....	338
33.5.16	Function: adv_logic.....	338
33.5.16.1	<i>add_client_stats</i>	338
33.5.17	Function: cpa_logic.....	338
33.6	log messages produced	339
33.7	Stopping skymds27	339
34	SKYMDS06	340
34.1	skymds06 Overview	340
34.2	Running skymds06	340
34.2.1	Command line parameters.....	340
34.2.2	Example command-line.....	341
34.3	Program Description	341
34.4	Program Structure	341
34.4.1	Sources files.....	341
34.4.2	Header Files.....	342
34.4.3	Database Tables.....	342
34.4.4	referenced environment variables.....	343
34.5	skymds06: Major Functions	343
34.5.1	Function: main.....	343
34.5.2	Function: initialise.....	343
34.5.3	Function: process_timed_events.....	343
34.5.4	Function: process_booked_call.....	344
34.5.5	Function: process_tms_reminder.....	344
34.5.6	Function: process_manual_reminder.....	344
34.5.7	Function: process_escalation.....	345
34.5.8	Function: process_fax_report.....	345
34.5.9	Function: process_email_report.....	345
34.5.10	Function: process_safety_<Company_Name>.....	346
34.5.11	Function: process_aparty_batch_reply.....	346
34.5.12	Function: update_timev_file.....	347
34.6	Log messages produced	347
34.7	Stopping skymds06	347
35	SKYMDS11R	348
35.1	skymds11r Overview	348
35.2	Running skymds11r	348
35.2.1	Command line parameters.....	348
35.2.2	Example command-line.....	348
35.3	Program Description	349
35.4	Program Structure	349
35.4.1	Sources files.....	349
35.4.2	Header Files.....	350
35.4.3	Database Tables.....	350
35.4.4	referenced environment variables.....	350

35.5	skymds11r: Major Functions.....	351
35.5.1	Function: main.....	351
35.5.2	Function: initialise.....	351
35.5.3	Function: process_sms_hist.....	351
35.5.4	Function: final_state.....	352
35.5.5	Function: process_smpp.....	352
35.5.6	Function: connect_and_bind_to_smsc.....	353
35.5.7	Function: check_enquire_<Company_Name>.....	353
35.5.8	Function: smpp_recv.....	353
35.5.9	Function: reply_to_smsc.....	354
35.5.10	Function: process_delivery_sm.....	354
35.6	log messages produced.....	354
35.7	Stopping skymds11r.....	354
36	CURSES OVERVIEW.....	356
36.1	Some important parts of a curses system.....	356
36.1.1	characters.....	356
36.1.2	WINDOW.....	356
36.1.3	subwindows.....	356
36.1.4	SCREEN.....	356
36.1.5	TERMINAL.....	356
36.2	Preparing the curses system.....	357
36.3	curses functions: a brief overview.....	357
36.3.1	Movement.....	357
36.3.2	Variations, forms of curses functions.....	357
36.3.3	Classes of curses functions.....	358
36.3.4	Implementing screen updates.....	358
36.3.5	Finalising curses and compiling.....	358
36.3.5.1	<i>program termination</i>	358
36.3.5.2	<i>compile time</i>	358
36.4	More Info On Curses.....	359
36.5	Curses Function List.....	359
36.5.1.1	<i>Add (Overwrite) Functions</i>	359
36.5.1.2	<i>Change Rendition Functions</i>	359
36.5.1.3	<i>Deletion Functions</i>	359
36.5.1.4	<i>Input from Keyboard to Window</i>	359
36.5.1.5	<i>Explicit Cursor Movement</i>	360
36.5.1.6	<i>Read Back from Window</i>	360
36.5.1.7	<i>Insert Functions</i>	360
36.5.1.8	<i>Print and Scan Functions</i>	360
36.5.1.9	<i>Function Keys</i>	360
37	<SYSTEM_NAME> CURSES UTILITIES.....	361
37.1	Functions in editscrn.c.....	361
37.1.1	Function: edit_scrn.....	361
37.1.2	Function: edit_report_screen.....	362
37.1.3	Function: mnt_get_input.....	362
37.1.4	The Other Functions.....	363
37.1.4.1	<i>display_screen</i>	363
37.1.4.2	<i>mnt_get_date_time</i>	363
37.1.4.3	<i>mnt_vali_date</i>	363
37.1.4.4	<i>mnt_get_invis_string</i>	363
37.2	Utility curses function in utils.h.....	363
38	EXAMPLES FROM A <SYSTEM_NAME> CURSES APPLICATION.....	364
38.1.1	Declare Fundamental Entities.....	364
38.1.2	Structure of a screen.....	364
38.1.3	Prepare the screen.....	364

38.1.4	Make the screen appear.....	365
38.1.5	Process the user's interactions.....	365
39	BULK SMS SPECIFICATION.....	366
39.1	Overview.....	366
39.2	SMS Functional requirements.....	366
39.2.1	Paging via Email.....	366
39.2.2	'B' Party SMS via Email.....	366
39.2.3	'A' Party SMS via Email.....	366
39.2.4	'A" Party Bulk SMS.....	366
39.3	System Architecture / Modules.....	366
39.3.1	Corporate Mail Gateway.....	367
39.3.2	sendmail.....	367
39.3.3	<System_Name> Email Gateway.....	368
39.3.3.1	<i>Initialization.....</i>	368
39.3.3.2	<i>Automatic Blocking.....</i>	368
39.3.3.3	<i>Processing of Email.....</i>	369
39.3.3.4	<i>Gateway Log File.....</i>	369
39.3.4	Reply Email.....	370
39.3.5	Email Blocking Maintenance.....	370
39.3.6	'B' Party Paging / SMS.....	371
39.3.7	'A' Party SMS.....	372
39.3.7.1	<i>Provisioning.....</i>	372
39.3.7.2	<i>Authentication.....</i>	372
39.3.7.3	<i>Real Time Processing.....</i>	373
39.3.7.4	<i>Bulk Processing.....</i>	373
39.3.7.5	<i>Format Of Text File Attachment.....</i>	374
39.4	Examples.....	374
39.4.1	Example 1.....	374
39.4.2	Example 2.....	374
39.4.3	Example 3.....	375
39.4.3.1	<i>Priority Queue processing.....</i>	375
39.4.3.2	<i>Capture of CDR's.....</i>	376
39.5	Special SMS Functions.....	378
39.6	Basic Bulk-SMS data Definitions.....	379
40	<SYSTEM_NAME> CODING NOTES.....	380
40.1	Each program has a port.....	380
40.2	Global and External variables.....	380
40.3	Return Types.....	380
40.4	Signatures.....	380
40.5	makefiles.....	380
40.5.1	Location.....	380
40.5.2	Header file inclusion.....	380
40.6	Hard coded variables, variable names.....	381
40.7	Program independence.....	381
40.8	Database tables.....	381
40.8.1.1	<i>Data structures.....</i>	381
40.8.1.2	<i>Management.....</i>	381
40.9	Grouping, handling maintenance functions.....	382
40.9.1	A call graph example.....	382
40.9.2	The levels.....	382
40.9.3	An extension of this Naming Pattern.....	382
40.10	initialise functions.....	383
<SYSTEM_NAME> CODE FILE EXAMPLE: SKYMD06.....		384

*ONLY*for personal, private use. Do not retain after use; do **NOT** distribute.
*ONLY*for use in association with hiring Stephen McGregor as a technical writer.

Table of Contents: Tables

Table 1: skymds00: Source Files.....	39
Table 2: skymds00: Header Files.....	39
Table 3: skymds00: Database Tables.....	40
Table 4: skymds00: Environment Variables.....	40
Table 5: skymds00: Base Menu actions.....	41
Table 6: skymds00: File Menu actions.....	42
Table 7: skymds00: Report Menu actions.....	43
Table 8: skymds00: Customer Menu actions.....	44
Table 9: skymds00: Customer Menu: Sub-menus.....	45
Table 10: skymds00: Utilities Menu actions.....	45
Table 11: skymds01: Source Files (1).....	46
Table 12: skymds01: Source Files (2).....	47
Table 13: skymds01: Source Files (3).....	48
Table 14: skymds01: Database Tables.....	49
Table 15: skymds01: Modules.....	49
Table 16: skymds01: Environment Variables Used.....	50
Table 17: skymds01: Key Short-cuts(1).....	53
Table 18: skymds01: Key Short-cuts (2).....	55
Table 19: skymds01: Key Short-cuts (3): Control keys.....	59
Table 20: skymds01: Contact codes & functions called.....	61
Table 21: skymds01: Pager number entry options.....	62
Table 22: skymds01: Control key-press table.....	62
Table 23: skymds01: Master key-press table.....	64
Table 24: skymds01: general paging key-press options.....	65
Table 25: skymds01 Advanced Features: form processing database tables.....	66
Table 26: skymds01 Advanced Features: Search database tables used.....	67
Table 27: skymds01 Advanced Features: call-patch database tables used.....	68
Table 28: skymds01 Advanced Features: Advisory database tables used.....	69
Table 29: skymds01 Advanced Features: ‘Myer’ Customized Forms functionality.....	70
Table 30: skymds01 Advanced Features: ‘Myer’ Customized Forms functionality (other).....	71
Table 31: skymds01 Advanced Features: ‘Safety <Company_Name>’ functionality.....	71
Table 32: skymds01 Advanced Features: ‘<Company_1_Name>’ (‘Startel’) functionality.....	71
Table 33: skymds01 Advanced Features: ‘V.I.P.’ Service codes.....	72
Table 34: skymds01 Advanced Features: ‘V.I.P.’ Operation codes.....	72
Table 35: skymds01 Advanced Features: ‘V.I.P.’ Programs and Functionality.....	72
Table 36: skymds01 Advanced Features: maintenance menu items and functions.....	73
Table 37: skymds01 Advanced Features: Escalation Acknowledgment (Events): database tables used.....	75
Table 38: skymds01 Advanced Features: Reference Numbers: Database tables used.....	76
Table 39: skymds01 Advanced Features: Reminders: Database tables used.....	76
Table 40: Functionality of skymds01? programs.....	79
Table 41: skymds01a: Message Retrieval: Database tables used.....	79
Table 42: skymds01a: Message Retrieval Key-press Functionality.....	80
Table 43: skymds01b: Booked Pages: Database tables used.....	81
Table 44: skymds01d: Pager Holder Name Maintenance: Database tables used.....	83
Table 45: skymds01e: Follow Me Maintenance: Database tables used.....	84
Table 46: skymds01f: Temporary Alternative Pager: Database tables used.....	86
Table 47: skymds01g: Group Paging Maintenance: Database tables used.....	87
Table 48: skymds01l: Temporary Messages: Database tables used.....	88
Table 49: skymds01f: Pager Details: Database tables used.....	90
Table 50: skymds08: Command-line Parameters.....	93
Table 51: skymds08: Source Files.....	94
Table 52: skymds08: Header Files.....	94
Table 53: skymds08: Database tables used.....	95
Table 54: skymds08: Environment Variables Referenced.....	95
Table 55: skymds08: get_valid_message() pre-processing.....	96
Table 56: mdsemgw: Command-line Parameters.....	101
Table 57: mdsemgw: Source Files.....	102
Table 58: mdsemgw: Header Files.....	102
Table 59: mdsemgw: Environment Variables.....	102

Table 60: mdsemgw: Environment Variables.....	102
Table 61: mdsem01: Command-line Parameters.....	107
Table 62: mdsem01: Source Files.....	108
Table 63: mdsem01: Header Files.....	108
Table 64: mdsem01: Database Tables Used.....	108
Table 65: mdsem01: Environment Variables Referenced.....	109
Table 66: mdsem02: Command Line Parameters.....	113
Table 67: mdsem02: Source Files.....	114
Table 68: mdsem02: Header Files.....	114
Table 69: mdsem01: Database Table used.....	114
Table 70: mdsem01: Environment Variables.....	115
Table 71: mdsem03: Command-line Parameters.....	121
Table 72: mdsem03: Source Files.....	123
Table 73: mdsem03: Header Files.....	124
Table 74: mdsem03: Database Table used.....	124
Table 75: mdsem01: Environment Variables.....	124
Table 76: mdsms01: Command Line Parameters.....	129
Table 77: mdsms01: Source Files.....	130
Table 78: mdsms01: Header Files.....	130
Table 79: mdsms01: Database Tables.....	131
Table 80: mdsms01: Environment Variables.....	131
Table 81: skymds96a: Command-line parameters.....	133
Table 82: skymds96a: Source Files.....	134
Table 83: skymds96a: Header Files.....	134
Table 84: skymds96a: Database Tables used.....	135
Table 85: skymds96a: Environment Variables.....	135
Table 86: skymds02: Command-line Parameters.....	139
Table 87: skymds02: Source Files.....	140
Table 88: skymds02: Header Files.....	140
Table 89: skymds02: Database Tables.....	141
Table 90: skymds02: Environment Variables.....	141
Table 91: skymds02: Handling Erroneous PET Transmissions.....	142
Table 92: skymds70: Command-Line Parameters.....	150
Table 93: skymds70: Source Files.....	151
Table 94: skymds70: Header Files.....	151
Table 95: skymds70: Database Tables.....	152
Table 96: skymds70: Environment Variables.....	152
Table 97: skymds18: Command-Line Parameters.....	156
Table 98: skymds18: Source Files.....	157
Table 99: skymds18: Header Files.....	158
Table 100: skymds18: Database Tables.....	158
Table 101: skymds18: Database Tables.....	158
Table 102: skymds18: Functions used to send messages to SKY<COMPANY_NAME>.....	159
Table 103: History Data Availability.....	169
Table 104: Meaning of Primary History file fields.....	172
Table 105: Meaning of Secondary History file fields.....	172
Table 106: Meaning of Secondary History tags.....	173
Table 107: Meaning of CDR file fields.....	183
Table 108: More detailed CDR Matrix.....	185
Table 109: skymds00: File Menu actions.....	204
Table 110: skymds00: Customer Menu actions.....	224
Table 111: Customer Menu: V.I.P. Sub-menu.....	225
Table 112: Customer Menu: <Company_1_Name>/X Sub-menu.....	225
Table 113: Customer Menu: F.R.C. Sub-menu.....	225
Table 114: Customer Menu: Safety-<Company_Name> Sub-menu.....	226
Table 115: skymds00: Utilities Menu actions.....	227
Table 116: skymds20: Command-Line Parameters.....	237
Table 117: skymds20: Source Files.....	238
Table 118: skymds20: Header Files.....	238
Table 119: skymds20: Database Tables.....	239
Table 120: skymds20: Environment Variables Us.....	239
Table 121: skymds20: Important Data Types.....	241
Table 122: mdsalarm: Command-Line parameters.....	245

Table 123: mdsalarm: Source Files.....	246
Table 124: mdsalarm: Header Files.....	246
Table 125: mdsalarm: Header Files (continued).....	247
Table 126: mdsalarm: Database Tables.....	247
Table 127: mdsalarm: Environment Variables.....	247
Table 128: skymds23: Command-Line Parameters.....	254
Table 129: skymds23: Source Files.....	255
Table 130: skymds23: Header Files.....	255
Table 131: skymds23: Database Tables.....	255
Table 132: skymds23: Environment Variables.....	255
Table 133: skymds23: When to Not process a maintenance queue message.....	256
Table 134: skymds23: socket configuration.....	258
Table 135: skymds24: Command-line parameters.....	259
Table 136: skymds24: Source Files.....	260
Table 137: skymds24: Header Files.....	260
Table 138: skymds24: Header Files (Continued).....	261
Table 139: skymds24: Database Tables.....	262
Table 140: skymds24: Environment Variables.....	263
Table 141: skymds24: Updates and Update Functions.....	265
Table 142: skymds24: Updates and Update Functions (Continued).....	266
Table 143: skymds24: Client Statistics Handler actions.....	267
Table 144: skymds24: Destinations Master Handler actions.....	267
Table 145: skymds24: Facilities Handler actions.....	267
Table 146: skymds24: SAGRN Handler actions.....	268
Table 147: skymds24: Primary History Updates Handler actions.....	268
Table 148: skymds24: Secondary History Handler actions.....	268
Table 149: skymds24: Indial update handler actions.....	269
Table 150: skymds24: pager record update handler actions.....	269
Table 151: skymds24: pager statistics update handler actions.....	269
Table 152: skymds24: public holiday update handler actions.....	270
Table 153: skymds24: generic processor handler actions.....	270
Table 154: skymds24: reminder update handler actions.....	270
Table 155: skymds24: reminder parameter update handler actions.....	271
Table 156: skymds24: search record update handler actions.....	271
Table 157: skymds24: security code update handler actions.....	271
Table 158: skymds24:timed events update handler actions.....	272
Table 159: skymds24: generic function descriptions.....	272
Table 160: skymds28: Command-line Parameters.....	274
Table 161: skymds28: Source Files.....	275
Table 162: skymds28: Header Files.....	275
Table 163: skymds28: Database Tables.....	276
Table 164: skymds28: Environment Variables.....	276
Table 165: skymds28: Top-level History Request Handlers.....	277
Table 166: skymds28: non-skymds28 programs using history functions.....	279
Table 167: skymds28: Specialised Primary History reading functions.....	280
Table 168: skymds03: Command Line Parameters.....	285
Table 169: skymds03: source files.....	287
Table 170: skymds03: header files.....	288
Table 171: skymds03: database tables.....	288
Table 172: skymds03: message handling functions.....	289
Table 173: skymds03: message delivery response actions.....	290
Table 174: skymds03: message handling functions.....	291
Table 175: skymds03: message handling functions.....	292
Table 176: skymds25: Command Line Parameters.....	303
Table 177: skymds25: Source Files.....	304
Table 178: skymds25: Header Files.....	304
Table 179: skymds25: Header Files (continued).....	305
Table 180: skymds25: Database Tables.....	305
Table 181: skymds25: Environment Variables.....	305
Table 182: skymds25: Maintenance Queue Message Handlers.....	306
Table 183: skymds25: Important Flags and Constants.....	307
Table 184: skymds25: <System_Name> to SKY<COMPANY_NAME> trans_type mappings.....	307
Table 185: skymds10: Command Line parameters.....	311

Table 186: skymds10: major behaviors and function calls.....	312
Table 187: skymds10: Source Code files.....	312
Table 188: skymds10: Header Files.....	313
Table 189: skymds10: Database Tables.....	313
Table 190: skymds10: Referenced Environment Variables.....	313
Table 191: skymds10: major behaviours and function calls [repeated].....	314
Table 192: skymds10: Standard I/O port control characters and meanings.....	317
Table 193: skymds10: Handling of standard I/O port control characters.....	318
Table 194: skymds10: non-message TNPP responses: processing.....	321
Table 195: skymds10: non-message TNPP responses: ignore.....	321
Table 196: skymds11t: Command Line parameters.....	323
Table 197: skymds11t: Source Files.....	324
Table 198: skymds11t: Header Files.....	324
Table 199: skymds11t: Database tables.....	324
Table 200: skymds11t: Environment Variables.....	325
Table 201: skymds27: ‘Contact’ types and Descriptions.....	329
Table 202: skymds27: Command Line Parameters.....	329
Table 203: skymds27: Source Files.....	330
Table 204: skymds27: Source Files (continued).....	331
Table 205: skymds27: Header Files.....	331
Table 206: skymds27: Header Files (continued).....	332
Table 207: skymds27: Database Tables.....	332
Table 208: skymds27: Environment Variables.....	332
Table 209: skymds27: Contact types, codes, and handlers.....	334
Table 210: skymds06: Types of timed events.....	341
Table 211: skymds06: Command Line Parameters.....	341
Table 212: skymds06: Source Files.....	342
Table 213: skymds06: Header Files.....	343
Table 214: skymds06: Database Tables.....	343
Table 215: skymds06: Environment Variables.....	344
Table 216: skymds06: Timed Event Handlers.....	345
Table 217: skymds06: Fax Types.....	346
Table 218: skymds06: Email Types.....	346
Table 219: skymds06: Safety <Company_Name>: Database Tables Used.....	347
Table 220: skymds06: Safety <Company_Name>: Functions Called.....	347
Table 221: skymds11r: Command Line parameters.....	349
Table 222: skymds11r: Source Files.....	350
Table 223: skymds11r: Header Files.....	351
Table 224: skymds11r: Database Tables.....	351
Table 225: skymds11r: Environment Variables Used.....	351
Table 226: curses: Add (Overwrite) Functions.....	360
Table 227: curses: change rendition functions.....	360
Table 228: curses: deletion functions.....	360
Table 229: curses: input functions.....	360
Table 230: curses: movement functions.....	361
Table 231: curses: read-from-window functions.....	361
Table 232: curses: insert functions.....	361
Table 233: curses: print functions.....	361
Table 234: curses: function keys.....	361
Table 235: curses: <System_Name> custom functions.....	362
Table 236: curses: <System_Name> screen structure.....	362
Table 237: Appendix 1 – SMS Definition: Email Blocking Table Definition.....	376
Table 238: Appendix 1 – SMS Definition: Email Queue Definition.....	377
Table 239: Appendix 1 – SMS Definition: A-Party Provisioning record format.....	377
Table 240: Appendix 1 – SMS Definition: A-Party record format.....	378

Introduction and Overview

1 Introduction

1.1 WHAT IS <SYSTEM_NAME>

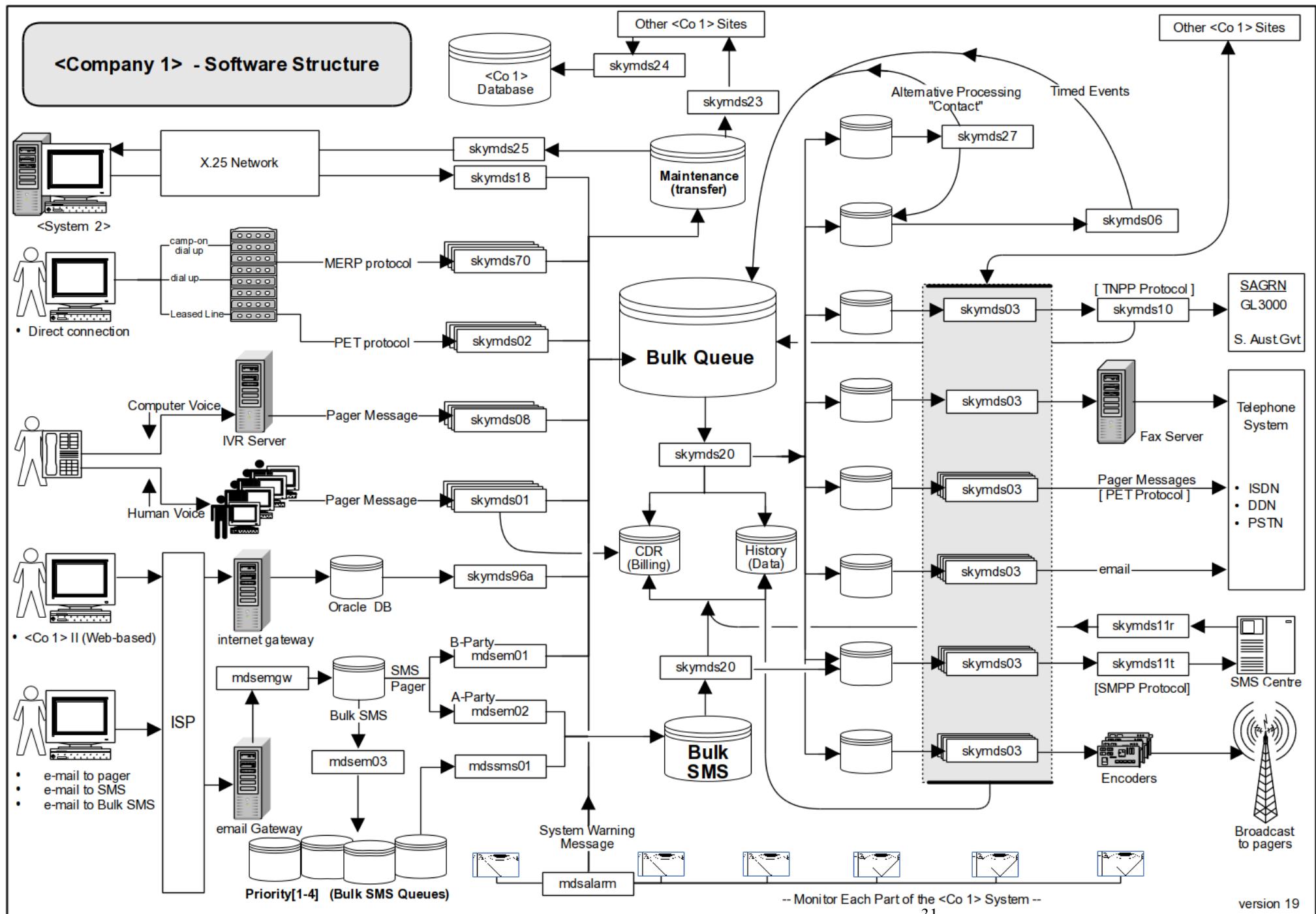
<System_Name> is a messaging system, delivering messages
from telephone, via a call centre or directly, from the internet and ‘special-delivery’ sources
to pagers, fax machines, mobile phones and email accounts.

Running on an HP 9000 server, with terminal servers, fax, internet, and history servers, call-centre hardware - plus a range of supporting devices - <System_Name> is large suite of specialist telecommunication applications, written in the C programming language and supported by an HP Unix (version 11.00) operating system. An Informix C-ISAM database is used for high-speed data storage and retrieval.

The five <System_Name> sites in Australia (Adelaide, Brisbane, Perth, Melbourne and Sydney) are uniformly configured:

- to collect messages:
 - a call centre, PABX, and XACOM phone-call processor,
 - an internet gateway
 - a modem ‘bank’
 - applications to run each of these devices and to manage collection of <System_Name> messages
- to process messages:
 - One or two HP 9000 servers running HPUX Unix version 11.00
 - The Informix C-ISAM database. The message queues are implemented as flat files.
 - TCP/IP and an X.25 networks.
 - applications to run each of these devices and to manage the processing of <System_Name> messages
 - a history collection process. All history older than seven days is moved to Melbourne.
- to send messages
 - encoders, amplifiers and transmitters for pager messages
 - fax and internet servers
 - connections to an SMS Gateway
 - PABX for forwarding pager messages for delivery by other telecommunications companies.
 - applications to run each of these devices and to manage transmission of <System_Name> messages

<System_Name> is a fast, integrated, and complete non-voice telecommunications solution. Born as a simple and robust paging system, twenty man-years of development has seen <System_Name> grow into the fully-featured, highly-optimised system it is today.



2 Programs Described – Brief Overview

See diagram 1.2, <System_Name> software structure

2.1 INPUT PROGRAMS

2.1.1 NORMAL PAGING INPUT PROGRAMS

2.1.1.1 **skymds00**

The ‘foundation’ <System_Name> program. skymds00 presents and manages the user logon / password screen, the base <System_Name> menu, and the base menu’s sub-menus. A copy of skymds00 remains running for the duration of each users’ <System_Name> session.

2.1.1.2 **skymds01**

The program that manages all the Call Centre operator screens, integrating the execution of the other programs that run behind the scenes

2.1.1.3 **skymds01 maintenance: skymds01a to skymds01p**

The message input system connects directly to a number of maintenance applications, skymds01a to skymds01p. More detailed and complete maintenance is available via the skymds00j sub-programs (*q.v.*).

2.1.1.4 **skymds08**

Interface to XACOM pbx/indial processor. skymds08 interacts with the IVR (interactive voice response) server to obtain pager message details from those who have had their pager-message call processed by a computer.

2.1.2 EMAIL / WEB TO PAGER, SMS AND BULK SMS MESSAGES

2.1.2.1 **mdsemgw**

Runs the email Gateway

2.1.2.2 **mdsem01**

Extracts the B-Party (normal) messages from the email message queue, posting them to the Bulk Queue

2.1.2.3 **mdsem02**

Extracts the A-Party (Sender Pays) messages from the email message queue, posting them to the Bulk Queue

2.1.2.4 **mdsem03**

Processes those emails that are to become Bulk-SMS messages, generating up to 50,000 SMS messages, storing them in one or more of four “priority” (Bulk SMS) queues.

2.1.2.5 **mdsms01**

Extract SMS messages from the four “priority” (Bulk SMS) queues, transferring them to the Bulk Queue.

2.1.3 FROM <SYSTEM_NAME> II (WEB-BASED) TO <SYSTEM_NAME>

2.1.3.1 **skymds96a**

Transfers messages from <System_Name> II’s message queue to the <System_Name> bulk queue for delivery via <System_Name> systems.

2.1.4 DIRECT CONNECTION TO <SYSTEM_NAME>

2.1.4.1 **skymds02**

Processes direct connections transferring messages in the PET format.

2.1.4.2 **skymds70**

Processes direct connections transferring messages in the (<Company_Name> Proprietary) MERP format.

2.1.4.3 **skymds18**

Used to transfer messages from <Company_Name>’s SKY<COMPANY_NAME> billing system to <System_Name>’s bulk queue.

2.2 CENTRAL PROCESSING AND MAINTENANCE PROGRAMS

2.2.1 THE MAIN DATA REPOSITORIES

2.2.1.1 The Bulk Queue

All messages in the <System_Name> system pass through the bulk Queue.

2.2.1.2 The Distribution Queues

Separate queues holding messages for processing by each of the delivery mechanisms.

2.2.1.3 The history Server, History Data

A detailed record of each message that passes through the <System_Name> system is stored in one or more history files.

2.2.1.4 CDR – Call Detail Records

Billing Data

2.2.2 MAINTENANCE OF THE <SYSTEM_NAME> SYSTEM

2.2.2.1 skymds00j

Maintain the configuration of each indial / client in <System_Name>

2.2.2.2 <System_Name> “file” menu programs

Programs that manipulate one or more <System_Name> configuration files

2.2.2.3 <System_Name> “Report” menu programs

Programs that generate and/or prepare a report

2.2.2.4 <System_Name> “customer” menu programs

Custom solutions prepared by <Company_Name> for particular clients.

2.2.2.5 <System_Name> “Utilities” menu programs

The <System_Name> utilities that need to be run through a GUI.

2.2.3 PROCESSING APPLICATIONS

2.2.3.1 skymds20

Extracts messages from the Bulk Queue for delivery to distribution queues

2.2.3.2 mdsalarm

Monitors <System_Name> for error conditions, and sends alarm messages appropriately.

2.2.4 DATA REPLICATION TO ALL <SYSTEM_NAME> SITES – MAINTENANCE QUEUE

2.2.4.1 skymds23

Sends messages through <Company_Name>’s LAN/WAN replicating each and every database change to all the other <System_Name> sites

2.2.4.2 skymds24

Receives the messages sent by skymds23 over the LAN and updates the local <System_Name> database.

2.2.4.3 skymds28

The history server application, that transparently accesses history data stored on non-local <System_Name> machines.

2.3 OUTPUT / DELIVERY PROGRAMS

2.3.1 THE DESTINATION SYSTEMS

- Paging
- SMS messaging – SMS Centres
- Forwarded Pager Messages (PET Protocol)
- Faxes
- SAGRN – South Australian Government Radio Network

2.3.2 DELIVERY TO EXTERNAL SYSTEMS: SKYMDS03

skymds03, using various modules, transfers messages from the bulk queue to the various systems

2.3.2.1 skymds03 sub-modules

A number of separately coded modules that give skymds03 the ability to deliver disparate types of messages.

2.3.3 OTHER EXTERNAL DELIVERY PROGRAMS

2.3.3.1 skymds25

Sends updates to the SKY<COMPANY_NAME> system on the Wang. The partner program to skymds18.

2.3.3.2 skymds10

Interface to the South Australian Government Radio Network (SAGRN) system. Uses “T.N.P.P” protocol.

2.3.3.3 skymds11t

Transmits SMS messages to the SMS Centres of other Telecommunication companies

2.3.4 IN THROUGH THE OUT-DOOR: INPUT SYSTEMS THAT CO-OPERATE WITH OUTPUT APPLICATIONS

2.3.4.1 skymds27

Manages and generates messages for alternative processing. Referred to as the “Contact” system.

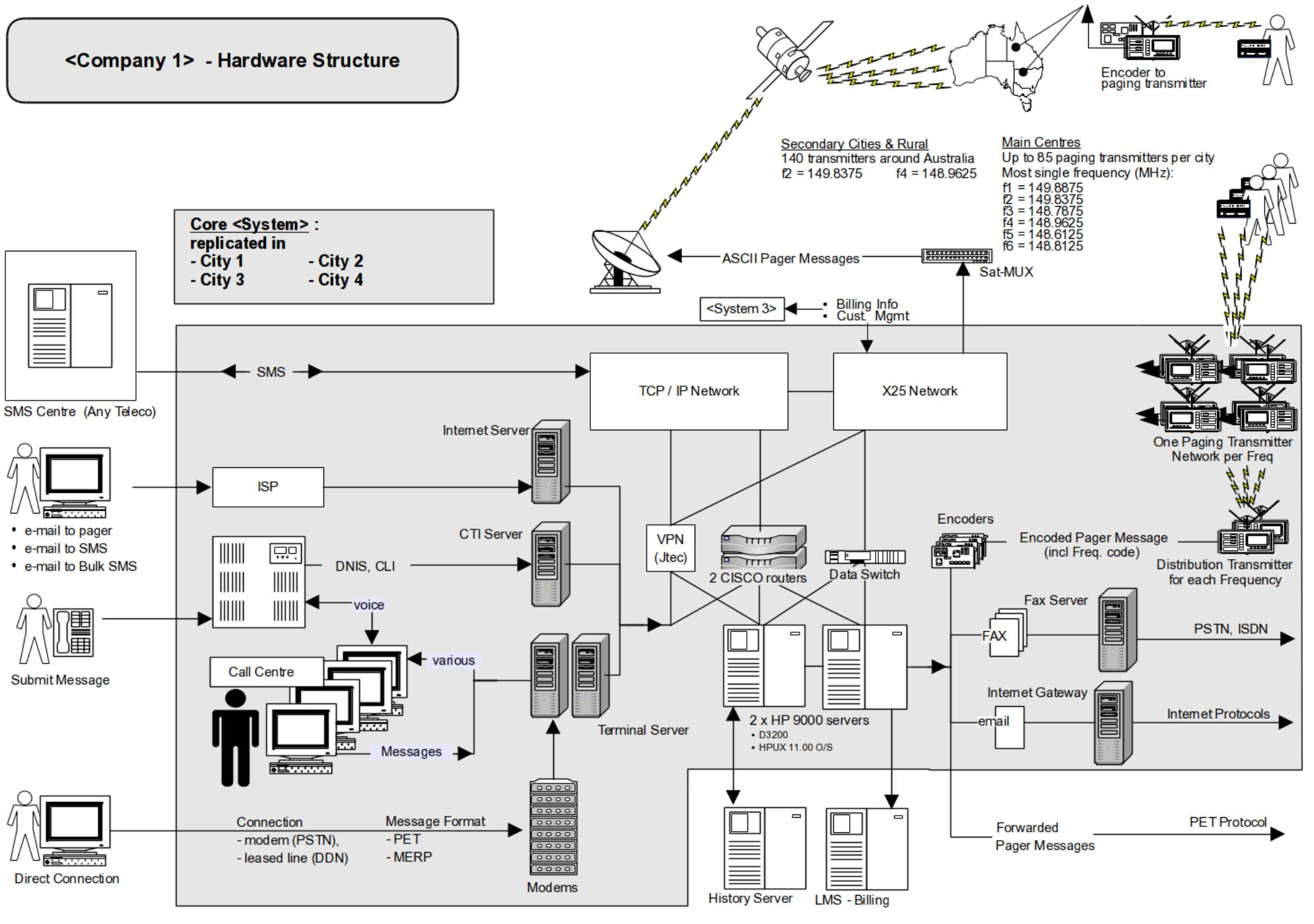
2.3.4.2 skymds06

skymds06 generates timed events, e.g. pager messages automatically sent at the same time each day.

2.3.4.3 skymds11r

Receives sent-SMS details from external SMS Centres.

<Company 1> - Hardware Structure



<System_Name>: Input Programs

ONLY for personal, private use. Do not retain after use; do NOT distribute.
ONLY for use in association with hiring Stephen McGregor as a technical writer.

3 skymds00

3.1 SKYMD00 OVERVIEW

skymds00 presents and manages the:

- <System_Name> logon-on screen
- username / password validation
- <System_Name> base menu and all sub-menu presentation
- The execution of the appropriate program for the menu selection made by the user.

Note that a single copy of skymds00 stays running during the entire session of each user, regardless of what subprograms they run. A typical Call Centre Operator will (automatically) execute skymds01 (answering calls), running it until their lunch break, end of the day, or they collapse – depending on how hard the slave drivers are whipping them that day. The copy of skymds00 stays in the background, unused, during this time.

3.2 RUNNING SKYMD00

The user / operator / supervisor runs the script start.00, which simply calls skymds00.

3.2.1 COMMAND LINE PARAMETERS

skymds00 only takes one command-line parameter:

parameter	Description
'-t' < comms port >	Used to register the application against a port in the mdspports database table

3.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute skymds00 at one <System_Name> site is:

```
$MDS_BIN_DIR/skymds00 2>$LOG_SCR
```

skymds00 is simply run, with the stderr stream being copied to the device \$LOG_SCR, defined in start.env to be /dev/tty1p0, a terminal (?) somewhere.

3.3 PROGRAM DESCRIPTION

skymds00 runs the ‘base’ <System_Name> screens: login, and skymds01 or base menu plus submenus. Most users, after login, will be transferred straight to skymds01 to receive calls. Administrators, programmers and back-office support personnel will be presented with the base menu and submenus

Taking this view that it performs these tasks separately, the basic structure is:

3.3.1.1 Login window

After calling initialise() skymds00 uses get_valid_login() to display the login window, collect and process the supplied password. This is actually quite a long process.

3.3.1.2 skymds01 – take incoming calls

Most users will be transferred straight to the skymds01 program after login, so they (being Call Centre Operators) can start to receive incoming calls.

3.3.1.3 base menu

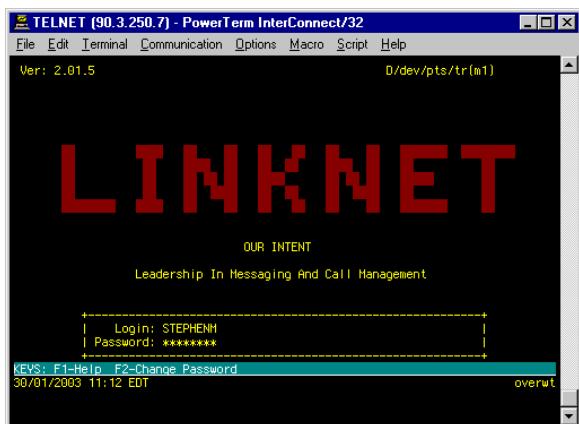
For the other users – administrators , programmers and back-office support personnel – the main_menu() function is called and they are presented with the base menu screen. rev_select() (a utility in utils.h) is used to gather the user’s menu choice which either calls a further *_menu() function (file_menu(), report_menu(), customer_menu(), or utilities_menu()) or a specialised program (skymds01 or skymds00j).

3.3.1.4 Submenus

A detailed listing follows, but the general pattern is:

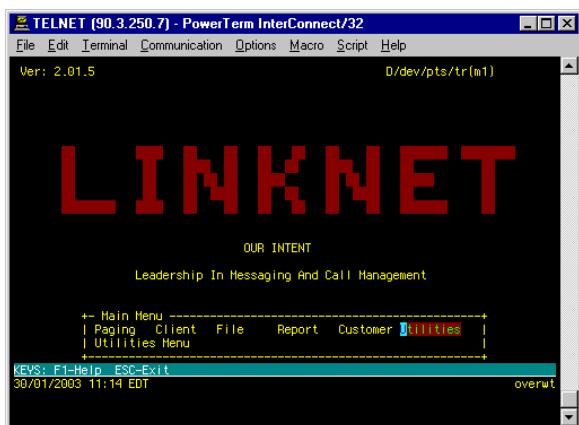
Menu choice	executes / calls	Description
Paging	skymds01	Manages incoming calls. See the (extensive) skymds01 documentation.
Client	skymds00j	Maintains client info, esp. contacts. See the skymds00j documentation
File	file_menu()	presents ‘file’ menu choices. Each choice spawns a program.
Report	report_menu()	presents ‘report’ menu choices. Each choice spawns a program.
Customer	customer_menu()	presents ‘customer’ menu choices. Each choice spawns a program.
Utilities	utilities_menu()	presents ‘utilities’ menu choices. Each choice spawns a program.

3.4 GUI: SAMPLE “SCREEN DUMPS”



3.4.1.1 Login screen

The first screen a user is presented with when entering <System_Name>. After entering a username and password the get_valid_login() processes and optionally approves the user.



3.4.1.2 Base menu screen

main_menu() displays the main <System_Name> menu. Using the rev_select() curses utility function to process the user’s menu selection it calls the appropriate program or function as detailed in the ‘submenu’ table above.



3.4.1.3 Sub-menu

The four sub-menu functions (the other two choices spawn executables) each present a menu from which, obviously, the user can make a further choice. Each choice spawns a curses-based application.

This screen dump is of the “utilities” sub-menu, activated by the utilities_menu() function (*q.v.*)

3.5 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds00.

3.5.1 SOURCES FILES

Source File	Description
editscrn.c	The routines for writing and reading from/to the Call Centre operators' (text) screens
exec_prog.c	A routine to execute (spawn) a program
form_path.c	Make a file-path from an environment variable and a filename
gensubs.c	Contains various general-purpose routines that are separately compiled.
help.c	Context-sensitive help for Call Centre Operators' screens (for skymds01)
iserror.c	A function to turn a C error message into a human-readable string
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
logerr.c	Simple function to dump an error message to standard error (stderr)
mdsginit.c	A number of functions, which perform commonly, used initialisation code segments.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the ports file.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdstmrtn.c	Contains functions, which perform operations to do with time.
scrdump.c	Dump the screen to a printer file
security.c	Various Security Routines.
skymds00.c	SKYPAGE PC logon & menu
spawn.c	Spawns off a child process and (optionally) waits for its completion.
stubs.c	Write (screen dump) a Call Centre Operator's (skymds01) screen to a file
utils.c	Various general-purpose MDS routines.

Table 1: skymds00: Source Files

3.5.2 HEADER FILES

Header File	Description
ascii.h	Function keys and special chars.: PMS function key values.
editscrn.h	The structure <System_Name> uses to hold (curses) window descriptions
gentypes.h	User defined types and booleans
get_input.h	rev_select() edit flags: editing manifests
gettime.h	C time field definitions.: C time field definitions
help.h	help functions
helpcodes.h	help context codes
macros.h	User defined macros.: definitions & macros for common functions
mdsbtrc.h	Bulletin master file
mdscdr.h	CDR definitions file
mdsderec.h	Ports security master file
mdsflsz.h	Standard <System_Name> includes.: field length & definitions
mdsg1rec.h	GL3000 tnpp record: <System_Name> -> GL3000 CDR record layout
mdsg2rec.h	GL3000 Billing record: GL3000 -> <System_Name> CDR record layout
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsparam.h	parameters used by PMS system
mdsporec.h	Ports security master file: Ports Record
mdspsrc.h	Password master file
mdsserec.h	Security Group master file
mdstrc.h	MNTQ: maintenance queue file
security.h	Structure and defines for security groups

Table 2: skymds00: Header Files

3.5.3 DATABASE TABLES

Each of these header files contains (only) a C-struct that matched the structure of a database table used by skymds00.

Database Table	Header File	Description
mdsbull	mdsbtrec.h	Operator bulletin
mdsdestm	mdsderec.h	Destinations master
mdsgl01	mdsglrec.h	Record layout of message from <System_Name> to a TNPP site
mdsgl02	mdsg2rec.h	g13000 version 6.00a billing protocol
mdsline	mdslirec.h	Indial number information
mdspassw	mdspsrc.h	Password username
mdsports	mdsporec.h	Ports security
mdsqal.txt	mdstrrec.h	Inter processor transaction queue
mdssecur	mdsserec.h	Security codes

Table 3: skymds00: Database Tables

3.5.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description	Example
PORT_TYPE		
MDS_DATA_DIR		
MDS_MACHINE_ID		

Table 4: skymds00: Environment Variables

3.6 SKYMD00: START-UP AND LOGIN FUNCTIONS

3.6.1 FUNCTION: MAIN

After the usual initialisation main() runs an infinite loop, first prompting for a user login and password before passing control to either **skymds01** or the **main_menu()** function, depending on the user's security level. When the user exits they are returned to the login screen.

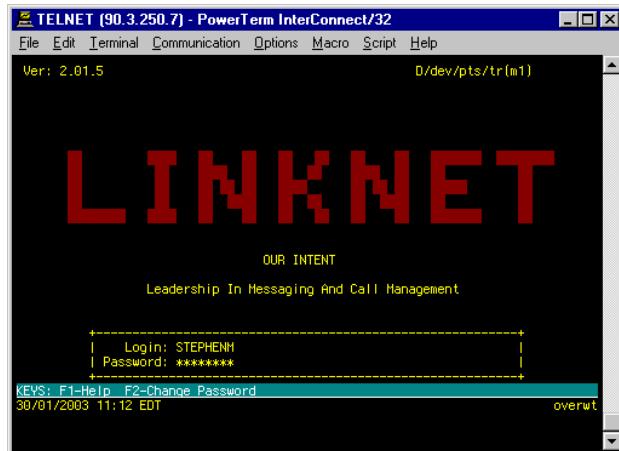
3.6.2 FUNCTION: INITIALISE

initialise sets up skymds00's run-time environment.

- first it parses the command-line, optionally reading the port into the mdspports database table.
- read in the environment variables used
- open a connection to the maintenance queue for inter-city updates.
- note in the mdspports database table that this instance skymds00 is running.
- open the port passed on the command-line as a communications port. It is not just being used as a tag – as it often is – in skymds00.
- deflect some signals.
- initialise the curses system and the base window
- attach to the security shared memory

3.6.3 FUNCTION: GET_VALID_LOGIN

- create the login 'box' ("window")
- prepare a record in the ports database table
- close unnecessary files
- reopen password, security files
- read the user's login name.
- read the users password

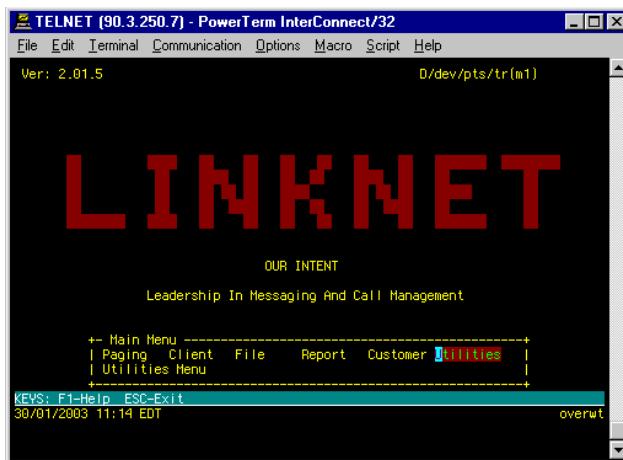


- given that the password is valid, get their security setting from security shared memory
- throw them out again if they have logged on incorrectly, or multiple times.
- handle the F2 – change password option
- close the password, security files.
- write the operator name to the mdspports record
- write the operator name, operator code, security-group index to the environment
- write the city-code, message prompt, port code, pager prefix, alternative pager prefix, and port type to the environment
- write the “general-line number” back to the mdspports ports database table.

3.7 SKYMD00: MAIN MENU

main_menu() displays the base <System_Name> menu and handles calls to the sub-menu programs or functions.

- first it constructs the menu members, testing the security shared memory before including each menu option.
- of course, it now displays the base menu to the viewing world.



- next it calls the rev_select() utility (in utils.h) to gather the user's selection from the menu.
- now based on the selection either handling function is called, or the appropriate program. See the table.

Menu choice	executes / calls	Description
Paging	skymds01	Manages incoming calls. See the (extensive) skymds01 documentation.
Client	skymds00j	Maintains client info, esp. contacts. See the skymds00j documentation
File	file_menu()	presents ‘file’ menu choices. Each of these spawns a program.
Report	report_menu()	presents ‘report’ menu choices. Each of these spawns a program.
Customer	customer_menu()	presents ‘customer’ menu choices. Each of these spawns a program.
Utilities	utilities_menu()	presents ‘utilities’ menu choices. Each of these spawns a program.

Table 5: skymds00: Base Menu actions

3.8 SKYMDS00: SUB-MENUS

The first two menu items call programs that have their own program-level documentation elsewhere in this document. skymds01, a huge program, has extensive documentation, while skymds00j, the client maintenance screen has documentation for itself and each of the 13 sub-programs / screens it calls.

The remaining four sub-menus have each of their items documented in the part of this manual dedicated to those menus. Here I will just briefly detail what each menu calls, and point the user to the appropriate fuller descriptions of the menu-items themselves.

3.8.1 MENU ITEM “PAGING” – SKYMD01

The Call Centre call-answering and message taking interface, skymds01, is spawned in result to either the “Paging” sub-menu choice, or, for Call Centre Operators, immediately on successful login, bypassing the base menu.

As per the table above: see the extensive documentation elsewhere.

3.8.2 MENU ITEM “CLIENT” – SKYMD00J

Choosing the menu item “Client” spawns the client maintenance screen and application skymds00j. This is also documented elsewhere.

3.8.3 SUB-MENU “FILE” - FUNCTION: FILE_MENU

The “file” menu choice calls file_menu(), which presents the menu (and manages spawning the process) which concern the overall set-up of the <System_Name> system.

- First the menu is constructed. These menu items and programs are hard coded (which is OK as the will seldom change) though which menu-items end up appearing for each user is individually determined based on the user’s security settings.
- The help context. Now, in order to understand the broad, far-reaching functionality available via this (‘file’) sub-menu, help is supplied:

```
+ Help -----+
 | FILE MAINTENANCE MENU
 |
 | The file menu is used by superusers and supervisors only. If you
 | are in here, you should know what you are doing. So alas there's
 | no constructive help here.
 +-----+
```

- The menu is displayed to the user.

Menu Item	Program	Comment
User/Password Maintenance	skymds00b	edit (add, delete, change) a password.
Operator Bulletin Maintenance	skymds00d	Edit the bulletin that appears on every user log-in.
Public Holiday Maintenance	skymds00p	The public holiday screen set-up / maintenance screen.
Function Code Maintenance	skymds39	assign access to parts of <System_Name> to groups of users.
Security Group Maintenance	skymds32	assign members and access attributes to groups
Audit Trail	skymds77	Display edits to ‘contact’ related database tables
Software Request Form Maintenance	skymds82	No longer relevant. Do not use.
Reminder Code Maintenance	skymds34	Set up details of pre-set reminder patterns
Email Blocking Maintenance	mdsemlblk	Stop users or domains using <System_Name> email services
Destination File Maintenance	mdsdestm	(Queues) Maintain details of <System_Name> message queues.

Table 6: skymds00: File Menu actions

Now, in an infinite loop, the user's choice(s) is/are processed.

- `rev_select()` reads the user's menu selection
- `exec_prog()` is used to run the appropriate subprogram. See the table above.
- `PORT_update_port_rec()` is used update the details in the mdsports database table to indicate that the current running process is no-longer whatever was spawned, but (back to) `skymds00` again.
- Once the user exits the memory allocated for the menu items is freed.

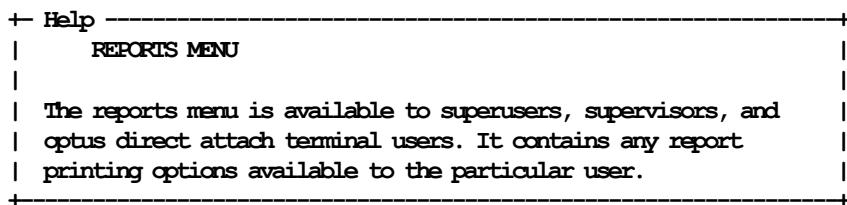
All done

3.8.4 SUB-MENU “REPORT” - FUNCTION: REPORT_MENU

This menu collects together a range of processes that generate reports on the state of the <System_Name> system and its processes (bet you never guessed that!)

This function is very similar to the file menu function just described.

- First the menu is constructed. These menu items and programs are hard coded (which is OK as the will seldom change) though which menu-items end up appearing for each user is individually determined based on the user's security settings.
- The help context is set. The help screen for the report menu (even more informative than for the file menu) is:



- The menu is then displayed top the user:

Menu Item	Program	Description
Set Printer	---	printer_select_menu(), which presents a list of <System_Name> printers.
Initialise Printer	---	Initialise the selected printer.
Report Generator	skymds46	Print out a report of data from the <System_Name> history. May also use one of: <ul style="list-style-type: none">▫ skymds46a – report based on event or system reference number▫ skymds46b – report based on contact ID▫ skymds46c – report based on pager number▫ skymds46d – report based on indial (phone) number
Pager Stats Report	skymds40	report on the number of calls to some pagers or account numbers.
Detailed Line Report	skymds47	‘line’ indial reporting. Uses skymds47a to access the database.
Answer Phrase Report	skymds48	all (yes, all) the answer phrases used, skymds48a accessing the database.
Batch Fax Setup	skymds19	Prepare batch faxes
Batch Fax List	skymds46e	Display a list of pending batch faxes.
Batch Email Setup	skymds19a	Prepare a batch email account.

Table 7: skymds00: Report Menu actions

Now, in an infinite loop, the user's choice(s) is/are processed.

- `rev_select()` reads the user's menu selection
- The sent printer and initialise printer menu-items are implemented directly within the `report_menu()` function.
- for the other menu choices `exec_prog()` is used to run the appropriate subprogram, except `skymds46e`, that is triggered via `spawn_program()` instead.
- `PORT_update_port_rec()` is used update the details in the mdsports database table to indicate that the current running process is no-longer whatever was spawned, but (back to) `skymds00` again.
- Once the user exits the memory allocated for the menu items is freed.

3.8.5 SUB-MENU “CUSTOMER” - FUNCTION: CUSTOMER_MENU

This menu is the interface to setting up and maintaining the custom solutions written by <System_Name> staff for the benefit of special <Company_Name> customers, these custom solutions handled in <System_Name> by implementing an ‘external database’. There are four external database custom solutions remaining in <System_Name>:

1. Myer Grace Bros Facilities Response Centre (FRC),
2. Safety <Company_Name>, a Personal Safety monitoring system,
3. <Company_1_Name> (cars) service request management
4. V.I.P. – A home-services franchise call handling

Now, and in the future, all new custom-solutions will (probably) be implemented within <System_Name> II, a web-based wrapper being built on top of the existing <System_Name>. These four remain in <System_Name>. The ‘customer’ menu provides the interfaces to the maintenance of each of these four services, the customer_menu() function following the same general procedure as the other menu functions.

- as before, the menu is first constructed, the user’s security level being tested before each item is added.
- Next the help context is added. Unfortunately the help for the reports menu is displayed instead of any help for this custom solution menu.
- Now the menu is displayed.

Sub-menu item	Calls:	Description
Trade/Service Code Maint		Prepare codes for a new custom solution
V.I.P. / V.I.P. Menu	vip_menu()	Home maintenance company set-ups
<Company_1_Name>/X	startel_menu()	<Company_1_Name> dealer information
Menu		
FRC Menu	frc_menu()	<Company_3_Name> company info
(<Company_3_Name>)		
Safety <Company_Name>	saf_menu()	Employee (safety) monitoring set-ups
Menu		

Table 8: skymds00: Customer Menu actions

- rev_select() reads the user’s menu selection.
- The menu-item text is then read to determine which sub-menu function (or skymds30) to call.
- The appropriate sub-menu is displayed:

Main Menu	Sub Menu	Program	Description
‘V.I.P. / V.I.P. Menu’			
	VIP Bulk Data Entry	cusvip02	Add delete new areas and services to the VIP Company
	VIP Database Maintenance	cusvip03	Read and change VIP data held by <Company_Name>
	VIP Database Report	cusvip06	Report based on entered data.
‘<Company_1_Name>/X Menu’			
	<Company_1_Name>/X	cussta02	Maintain the <Company_1_Name> dealer info
	Dealer Maint		
	<Company_1_Name>/X	cussta04	Simpler where-is-each-dealer info
	Location Maint		
‘FRC Menu (<Company_3_Name>)’			
	FRC Manager Maint	cusfrcfm	Maintain <Company_3_Name> manager info
	FRC Enquiry Type Maint	cusfrct	Maintain Enquiry Type info
	FRC Enquiry Code Maint	cusfrcen	Maintain Enquiry Code info
	FRC Site Maint	cusfrcsi	Maintain Site info
	FRC Enquiry/Site Maint	cusfrces	The relationship between location and enquiries
	FRC Trade Code Maint	cusfrctr	Maintain trade codes
	FRC Maintenance Code Maint	cusfrema	Maintain maintenance codes.
	FRC Service Provider Maint	cusfrcsp	Maintain service providers
	FRC Trade/Site Maint	cusfrcts	Maintain the Trade code – location relationships
	FRC Site/Cost Centre Maint	cusfrsc	Maintain the location – cost centre relationships
	FRC Escalation Rebuild	cusfrcer	Build message escalation rules.
‘Safety <Company_Name> Menu’			

Safety <Company_Name> Profile Screen	cussafpr	Information about the people being monitored
Safety <Company_Name> Service File	cussafsp	Prompts and set-up for the Safety <Company_Name> service
Safety <Company_Name> log file Enquiry	cussaflg	Search past Safety-<Company_Name> data
Report Generator	cussafrp	Produce a report on Safety <Company_Name> actions.

Table 9: skymds00: Customer Menu: Sub-menus

Each of these further sub-menu functions: vip_menu(), startel_menu(), frc_menu(), saf_menu() is written, and operates, in the same way as each of the menu function described so far. Depending on the sub-menu choice the program as displayed in the above table is called.

- PORT_update_port_rec() is used update the details in the mdsports database table to indicate that the current running process is no-longer whatever was spawned, but (back to) skymds00 again.
- Once the user exits the memory allocated for the menu items is freed.

3.8.6 SUB-MENU “UTILITIES” - FUNCTION: UTILITIES MENU

The “Utilities” menu choice calls utilities_menu(), giving access to various commonly used utilities. There are many more utilities available in the <System_Name> system, this is just a few of them.

- First the menu is constructed, as before, which menu-items appearing for each user is determined based on the user’s security settings.
- The help context is set, again displaying the help for the Reports menu
- The menu is displayed to the user.

Menu Item	Program	Description
Watch Queue (60 Seconds)	mdswq	Monitor all <System_Name> queues – update every minute
Watch Queue (5 Seconds)	mdswq	Monitor all <System_Name> queues – update every 5 seconds
Operator List Report	skymds33	List Call Centre Operators by city and security group
Terminal Monitoring	mdstymon	remotely monitor an operator’s terminal
Network Delay Monitoring	skymds78	view faxes sent by indial, fax number, or (original) pager number.
Network Connection maintenance	skymds79	Manage the (many) <System_Name> network connections.
Mobile Phone Maintenance	skymds65	Special information for special (e.g. Virgin / Optus) mobiles
Port Maintenance	skymds84	Static, fixed information on ports
Alarm Maintenance	skymds92	Set the testing frequency and destination of <System_Name> alarms.
Customer Program Maintenance	skymds88	When to run the custom solutions. (see ‘customer’ menu)
Fax Delivery Status	skymds78	Monitor which faxes will be and have been sent.

Table 10: skymds00: Utilities Menu actions

Now, in an infinite loop, the user’s choice(s) is/are processed.

- rev_select() reads the user’s menu selection
- Each choice triggers the program listed above, some with parameters
- PORT_update_port_rec() is used update the details in the mdsports database table to indicate that the current running process is no-longer whatever was spawned, but (back to) skymds00 again.
- Once the user exits the memory allocated for the menu items is freed.

4 skymds01

4.1 SKYMDS01 OVERVIEW

skymds01 manages a suite of programs that comprise the <System_Name> Call Centre operators' message entry system. Spawning by the skymds00 logon program, skymds01 presents Call Centre operators with message entry screens, from which it collects and manages the various message entry functionality.

skymds01 uses the Unix 'curses' system to display text-boxes, frames, input/output, overlapping windows and the like on the Call Centre operators' computer screens. Curses, the stone-age version of Windows dialogs, windows and other pop-ups, uses text characters to display text on a screen. For example, a curses dialog may a string of '-----' to draw a line. See the Unix man page 'curses_intro' and 'curses(5)' for a detailed overview.

This program, skymds01, is one of the oldest - started in 1989 - and most central <System_Name> programs; the vast majority of <System_Name> messages enter through this interface (via <System_Name> Call Centres). Further, skymds01's age and importance have lead to a huge amount of work being applied to this/these program(s). The largest (sets of) functions have been split off into sub-modules, many different programmers have made changes and improvements, and skymds01 has, over time, developed into a complicated labyrinth of code. May this documentation be your guide ...

4.2 RUNNING SKYMDS01

A Call Centre operator types "start.00" at the Unix command prompt. This simple script runs the skymds00 logon program to receive and process the user's logon name and password. Once a user's logon is validated skymds00 runs skymds01.

skymds00 calls exec_prog() (coded in exec_prog.c) to run skymds01. **exec_prog()** in turn calls **spawn_prog()**, a utility function for running arbitrary programs.

4.2.1 COMMAND LINE PARAMETERS

The only command line parameter is an optional "insert" or "overwt", indicating the insert or overwrite status of skymds01 upon startup.

4.3 PROGRAM DESCRIPTION

As described above, skymds01 supplies and manages the screens displayed to the Call Centre operators as they are taking and processing calls. It is entirely text based, using the common Unix curses (text) system to construct and display the screens.

4.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds01.

4.4.1 SOURCES FILES

Source File	Description
skymds01.c	Manages the Call Centre Operators' Message entry screens
alarmrtn.c	Generic alarm routine.
disp_pager.c	Display a pager number
editscrn.c	The routines for writing and reading from/to the Call Centre operators' (text) screens
exec_prog.c	A routine to execute (spawn) a program
fileopt.c	Open and close files as needed – optimise (minimise) number of open files
form_path.c	Make a file-path from an environment variable and a filename
gensubs.c	Contains various general-purpose routines that are separately compiled.
get_pager.c	Pager number input, holder name enquiry.
globvar.c	Functions to access the Global Variable shared memory used by multiple programs.
help.c	Context-sensitive help for Call Centre Operators' screens (for skymds01)

Table 11: skymds01: Source Files (1)

(continues ...)

Source File	Description
iserror.c	A function to turn a C error message into a human-readable string
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
logerr.c	Simple function to dump an error message to standard error (stderr)
maskpagr.c	Read pager file with line file driven masking.
mds01adv.c	Advisory handling routines used in skymds01.c
mds01ans.c	The answer phrase displayer used in skymds01.c
mds01con.c	Contact handling routines used in skymds01.c
mds01cpa.c	CALL PATCH calling module for skymds01.c
mds01edb.c	Module which initiates screen programs that processes a customer's External database.
mds01evt.c	Event number entry processing.
mds01frm.c	Forms handling routines used in skymds01.c
mds01ini.c	The skymds01.c initialisation routine.
mds01man.c	Various manual select routines used in skymds01.c
mds01rem.c	Reminder retrieval routines used in skymds01.c
mds01sch.c	SEARCH calling module for skymds01.c
mds01srn.c	System Reference Number entry processing.
mdsadv.c	Advisory routines
mdscdr.c	Contains various general-purpose routines that are separately compiled.
mdscomp.c	Company Details routines
mdsepa.c	CALL PATCH Screen routine called by other programs
mdscstats.c	Increment 'X-has-happened' count in a program statistics log file
mdsedb.c	Module which initiates screen programs that processes a customer's External database.
mdsemqmain.c	Generic message email queuing routine.
mdsfrm.c	Contains various general-purpose routines that are separately compiled.
mdsginit.c	A number of functions that perform commonly used initialisation code segments.
mdshisio.c	Generic history file handling routines.
mdskbld.c	Contains functions to build and unbuild pager keys, and set-up a dummy pager.
mdslog.c	Generic error logging.
mdsmnt.c	Maintenance (CONTROL) window routines used in skymds01
mdspol.c	Functions to attach, check and modify Public Holiday shared memory. Used by multiple programs.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurtn.c	Generic message queuing routine.
mdsrem.c	Functions to access the reminders queue, used by multiple programs
mdssch.c	SEARCH Screen routine called by other programs
mdssetup.c	Obtain set-up definitions from a text set-up file.
mdssmsmsg.c	Display a message from an SMS message
mdssmsrt.c	Contains various general-purpose routines that are separately compiled.
mdstamp.c	Date time stamp & sequence number generation
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions that perform operations to do with time.
mdsuniqu.c	Functions to get and update the Unique Number files
messent.c	The skymds01 message entry editor.
scrdump.c	Dump the standard screen 'stdscr' to a printer file
security.c	Various Security Routines.
spawn.c	Spawns off a child process and (optionally) waits for its completion.
stubs.c	Write (screen dump) a Call Centre Operator's (skymds01) screen to a file
utils.c	Various general-purpose MDS routines.

Table 12: skymds01: Source Files (2)

4.4.2 HEADER FILES

Header File	Description
acdmem.h	ACD specifics - (uses LIREC): shared memory layout
ascii.h	Function keys and special chars.: PMS function key values.
fastpath.h	"fastpath" maintenance functions
gentypes.h	User defined types and booleans
get_input.h	rev_select() edit flags: editing manifests
gettime.h	C time field definitions.: C time field definitions
globvar.h	Global shared memory structures
help.h	help functions
helpcodes.h	help context codes
macros.h	User defined macros.: definitions & macros for common functions
mdsadrec.h	(ADV) Advisory Master File
mdsaprec.h	A-Party Master file
mdscdr.h	CDR definitions file
mdscdrec.h	CDR data structure
mdsclrec.h	Client master file
mdscmrec.h	Company Details file
mdscorec.h	(CLT/INV/GRP) Contacts Master File
mdscprec.h	Call Patching Master File
mdscsrec.h	Client stats master file
mdscurec.h	Customer Program File
mdsderec.h	Ports security master file
mdsetrec.h	Escalation transaction master file
mdsfarec.h	Pager facilities master file
mdsflsz.h	Standard <System_Name> includes.: field len & definitions
mdsfmrec.h	(FRM) Forms Master File
mdsfstrec.h	Forms text File
mdshirec.h	History primary recs. file: New History logging file
mdshsrec.h	Secondary History file (audits)
mdsiprec.h	Info Pack master file
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsparam.h	parameters used by PMS system
mdsparec.h	Pager Master File
mdsphol.h	Public Holiday Shared Memory file
mdsphec.h	(TMS) Telephone Message Service
mdsporec.h	Ports security master file: Ports Record
mdspsec.h	Password master file
mdsqrec.h	Contacts Message queue file: Destination Message queue file
mdsrem.h	Reminders shared memory definition
mdsrrec.h	Reminders master file
mdssprec.h	Search Parameter master file
mdstmrec.h	Temporary message master file
mdsunqn.h	Unique No. function parameter struct
mdsunrec.h	Unique Number File
pagernum.h	Pager number entry manifests.
security.h	Structure and defines for security groups

Table 13: skymds01: Source Files (3)

4.4.3 DATABASE TABLES

Each of these header files contains (only) a C-struct that matches the structure of a database table.

Database Table	Header File	Description
mdsadv	mdsadrec.h	Advisory
mdsaparty	mdsaprec.h	mds A-Party record structure
mdscdrec	mdscdrec.h	Call details master
mdsacrec	mdscdrec.h	Alternative call details master
mdsclien	mdsclrec.h	Client master
mdsemdet	mdsemrec.h	Company details
mdsconta	mdscorec.h	Contacts master
mdscallp	mdscprec.h	Call patching master
mdsclsts	mdscsrec.h	Client stats master
mdscustp	mdsecurec.h	Customer program
mdsdestm	mdsderec.h	Destinations master
mdsestrn	mdsetrec.h	Escalation transaction master
mdsfacil	mdsfarec.h	Facilities – what special <System_Name> services can be used
mdsfmtxt	mdsfmrec.h	Forms text
mdsforms	mdsfmrec.h	Forms
hiccyymmdd	mdshirec.h	the 'logical' history
hsyymmdd	mdshsrec.h	History secondary storage
mdsinpak	mdsiprec.h	Info pack.
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsphone	mdsphtrec.h	Phone master
mdsports	mdsporec.h	Ports security
mdspassw	mdpsrec.h	Password username
mdsqxx.txt	mdsquarec.h	Destination queue s
mdsrem	mdsrrec.h	Reminders queue
mdsschp	mdssprec.h	Search parameter
mdstmesg	mdstmrec.h	Temporary message master
mdsuniqn	mdsunrec.h	Unique number

Table 14: skymds01: Database Tables

4.4.4 SKYMDS01 MODULES

Twelve major sub-modules are used by skymds01. These are briefly listed in the table that follows. Note that, technically, these are no different from any other source-code file compiled into skymds01. <System_Name> does not use shared dynamically <Company_Name>ed modules or precompiled libraries.

Module	Description
mds01adv.c	Advisory handling routines used in skymds01.c
mds01ans.c	The answer phrase displayer used in skymds01.c
mds01con.c	Contact handling routines used in skymds01.c
mds01cpa.c	CALL PATCH calling module for skymds01.c
mds01edb.c	Module which initiates screen programs that processes a customer's External database.
mds01evt.c	Event number entry processing.
mds01frm.c	Forms handling routines used in skymds01.c
mds01ini.c	The skymds01.c initialisation routine: <code>initialise()</code>
mds01man.c	Various manual select routines used in skymds01.c
mds01rem.c	Reminder retrieval routines used in skymds01.c
mds01sch.c	SEARCH calling module for skymds01.c
mds01srn.c	System Reference Number entry processing.

Table 15: skymds01: Modules

4.4.5 REFERENCED ENVIRONMENT VARIABLES

Environment variables are either:

- pre-existing in the computer's run-time environment
- added to the start-up environment from the start.env file
- exported to the environment in the start.mds start-up file itself

skymds01, run by skymds00, can utilise the first two of these three options.

Environment Variable	Description
MDS_MACHINE_ID	What server am I using – a proxy for what city am I in
MDS_DATA_DIR	Where is the data directory? Usually /var/mds/data/.
MDS_BIN_DIR	Where the executables are. Usually /opt/mds/bin/
OUR_TTY	The current terminal (display) device.
OPERATOR	The Operator Code
OPER_NAME	The Operator's name
TERM	The terminal type
SECURITY_INDEX	What Security group do I / does this user belong to
OUR_CITY	M=Melbourne, S=Sydney ...
PORT	The communications port being used. Something like /dev/ttyp8.
SWITCH_POS	What PABX line the operator is connected to
PORT_TYPE	r = remote, L= local ...
MDS_SHMIDS_DIR	The directory containing files holding the shared memory keys

Table 16: skymds01: Environment Variables Used

4.5 STANDARD FUNCTIONS

4.5.1 FUNCTION: MAIN

`main()` passes control to either `base_logic()` or `general_logic()` to implement skymds01's workload. `base_logic()` waits for a phone call from the PABX, while `general_logic()` immediately transfers the user to a pager-message entry screen

- first `main()` call `initialise()` to set up skymds01's environment, separately coded in the mds01ini.c file.

It is this `initialise()` function that displays the first <System_Name> base screen, though remember that both the logon and start menu screens are generated by skymds00, the program that spawns skymds01.

- handle signals. `setup_signal_catching()` performs the signal handling usually done in a <System_Name> `initialise()` function, i.e. catch and write a line to an error log.
- Next, the choice between `base_logic()` and `general_logic()` is made.

`check_security_level()` tests the security of the user. If restricted access paging is indicated, that is, a '1' in position 66² of the (group) security shared memory area indicated by the SECURITY_INDEX environment variable, or the environment variable PORT_TYPE is 'R', 'r', or 'S', the `general_logic()` (send a pager message) screen is opened, else the screen goes blank waiting to receive an incoming pager call. (`base_logic()`).

- `general_logic()` or `base_logic()` is called. Control remains in the chosen one of these functions until skymds01 terminates.
- `CDR_end_session()` is called, the long winded way to close the CDR (billing) data file.

Due to the unique naming system of CDR files, a call to a separate function in a separate source file (mdscdr.c) probably is justified.

- `cleanup()` is called

² SEC_RESTRICTED_PAGING == 66



4.5.2 FUNCTION: INITIALISE

coded in : mds01ini.c NB: not in skymds01.c
 called by : main.c
 raison d'être : does all the preparation for running skymds01.

initialise() operates in the following manner:

- A few signals are handled, probably to ensure that this initialisation function runs properly.

A separate function: setup_signal_catching(), handles the full signal handling for skymds01.

- A few environment variables are read in from the environment.

Only 5 of the 20 environment variables are read at this stage. These are the ones necessary to initialise the curses screen writing system. After the first curses window has been displayed the remainder of the initialisation function continues, so as to display a first screen as quickly as possible to the Call Centre operator.

- the ‘curses’ screen-drawing system is initialised
- the base, background window, and the status bar are drawn and displayed to the user.
- The command-line is parsed.

Only one, optional, parameter is expected on skymds01 command-line “insert” or “overwt”.

- Get the next 15 or so environment variables.
- Open all the database tables and files.

This one is interesting. The pager, destinations and facilities (special service permissions) database tables are opened. Next the password table is checked for the Call Centre operator’s details and closed. Now, 13 other, less central tables are added to “the optimizer [sic]”, a device that keeps open only a certain number of database tables at any time. Those not used for a while are quietly closed and re-opened when a request is made. With 100 or so copies of skymds01 running on any one server, the wisdom of such a move is clear.

- skymds01 attaches to the bulk queue text file.
- skymds01 attaches its to shared memory blocks:

- The shared memory key is read from a file in the **MDS_SHMIDS_DIR/acd** directory.

- The reminder³ shared memory block is obtained via the `REM_get_shared_mem()` function (in `mdsrem.c`). The decision whether to process reminders is made here, based, amongst other things, on the contents of this reminders shared memory block.
 - the global variables are similarly accessed via a shared memory pointer returned by the `get_globvar_shared_mem()` function (in `mdssetgl.c`)
 - The security shared memory block is obtained via `get_security_shared_mem()` (coded in `security.c`). The operator's security shared memory block is derived from this and the SECURITY_INDEX environment variable.
- Read the ‘destinations’: the two-letter queue identifiers mnemonics.
 - Open and connect to the maintenance queue

The maintenance queue is used for the transfer of messages between <System_Name> sites, in order to keep the databases synchronised.

- finally, set a flag so that, when retrieving an SMS message, it is possible to determine which operator or externally connected computer retrieved the message.

4.5.3 FUNCTION: GENERAL_LOGIC

coded in : skymds01.c
 called by : the skymds01 `main()` function, if the environment variable PORT_TYPE is ‘R’, ‘r’, or ‘S’
 raison d'être : a short-cut straight to paging for certain types of skymds01 users.

If ‘general_line’ (copied from the MDS_GENERAL_LINE environment variable) matches an indial pre-existing in the mdsline database, `general_logic()` passes control to `answer_logic()`. Else `general_logic()` (and thus skymds01) exits.

4.5.4 FUNCTION: BASE_LOGIC

coded in : skymds01.c
 called by : the skymds01 `main()` function, if the environment variable PORT_TYPE is not ‘R’, ‘r’, or ‘S’
 raison d'être : The post-`initialise()` entry point for skymds01. `base_logic()` retains control of skymds01 until user action or adventure brings the program to termination.

base_logic() operates in the following manner and order:

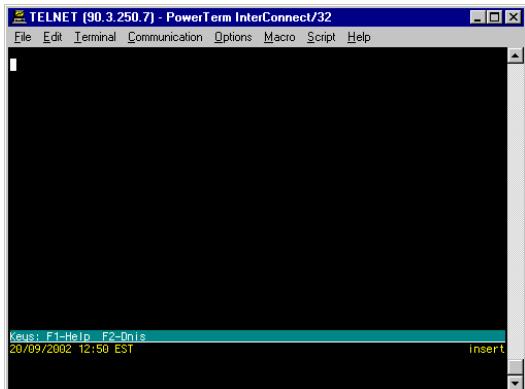
- Sets the help context, so that the appropriate help for this introductory screen is displayed upon a F1 key-press
- Start the infinite loop. Control will stay in this loop until expressly exited.
- Clear and re-draw the screen
- If the user should process reminders⁴, and there are reminders to process, display that the ‘F5’ key gets reminders

A skymds01 user will be able to process reminders if and only if:

- Their port_type is ‘L’. The port_type is read from the environment variable PORT_TYPE, with ‘L’ indicating that the current user is a “Local” user, i.e. directly connected to the <System_Name> machine via a terminal server, *and*
- in the <System_Name> password database table, the process_reminders field for this user is set to ‘Y’, *and*
- this is the main (‘master’) machine for the <System_Name> site, as indicated by the global shared block’s master_macth field (= ‘Y’)

³ Reminders are, well, reminders sent to Call Centre operators to do something at a particular time.

⁴ See footnote #2. Reminders are messages to Call Centre operators to undertake particular tasks at particular times.



What base_logic() looks like

This decision is made during the processing of the reminders shared memory block during skymds01 **initialise()** function.

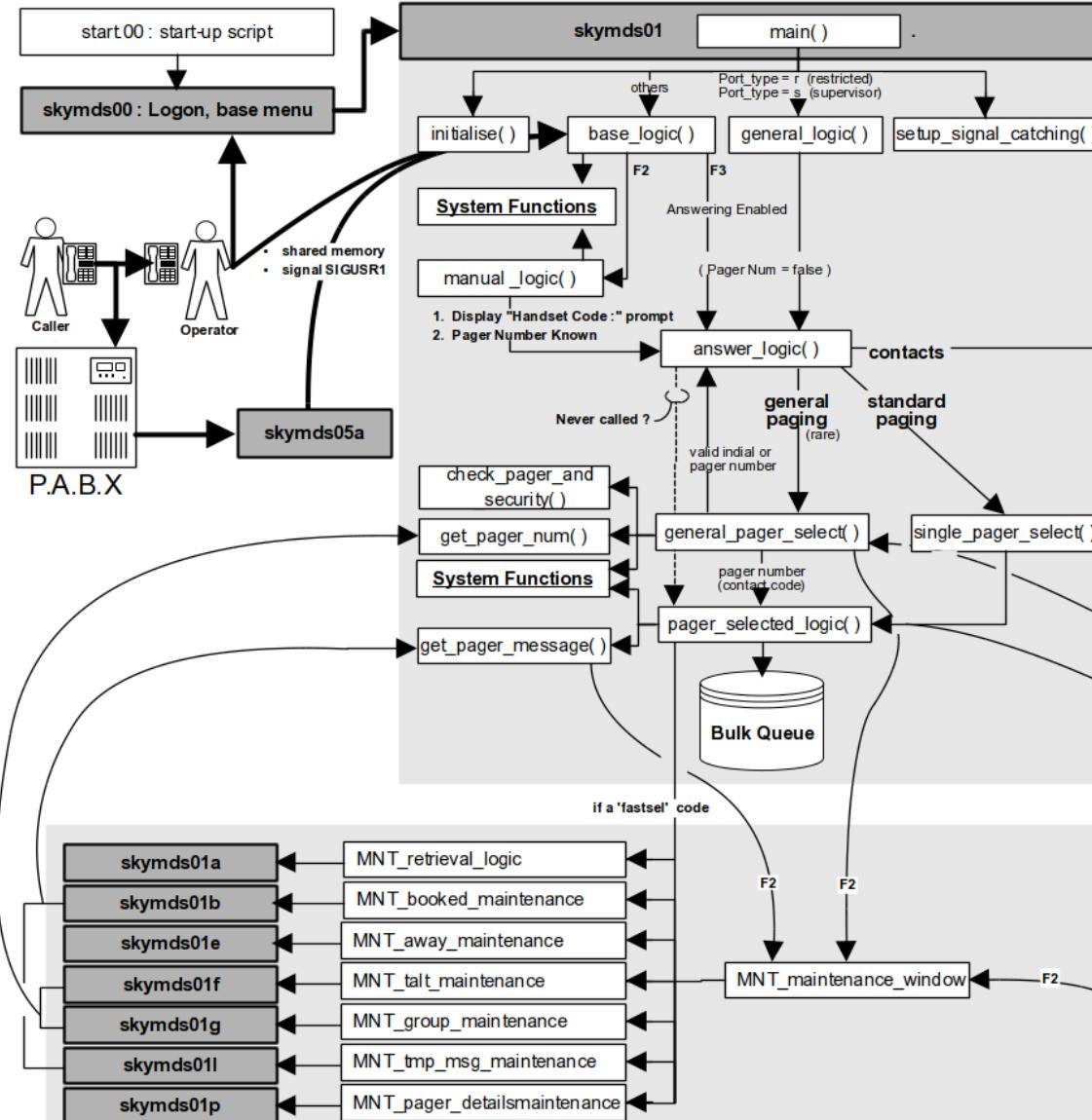
- See if we can get a call from the PABX
- Process the received call, or send a pager message without receiving a call, or whatever

Key Press	Action	Description
F2	send a pager message	call manual_logic() (q.v.)
F3	receive a call	<ul style="list-style-type: none"> - check that we are properly connected to shared memory. If not, then force the user to push the ESC key. - connect to the call in shared memory, or wait two seconds. If nothing happens then give up and try again. - Check that the incoming call indicates that it has a valid line-number - If it looks OK, transfer to answer_logic(), else call disconnect_call()
F4	acknowledge an escalation	call event_processing() . Escalation acknowledgements are “events”.
Shift-F4	process a reference number	call sys_ref_processing()
F5	process a reminder	call reminder_retrieval()
F8	Display operator details	call operator_details()
	Exit	If the user is a supervisor (including a programmer), the exit is immediate. Otherwise the Call Centre operator is asked to confirm that they wish to exit, after which they are exited from the system.

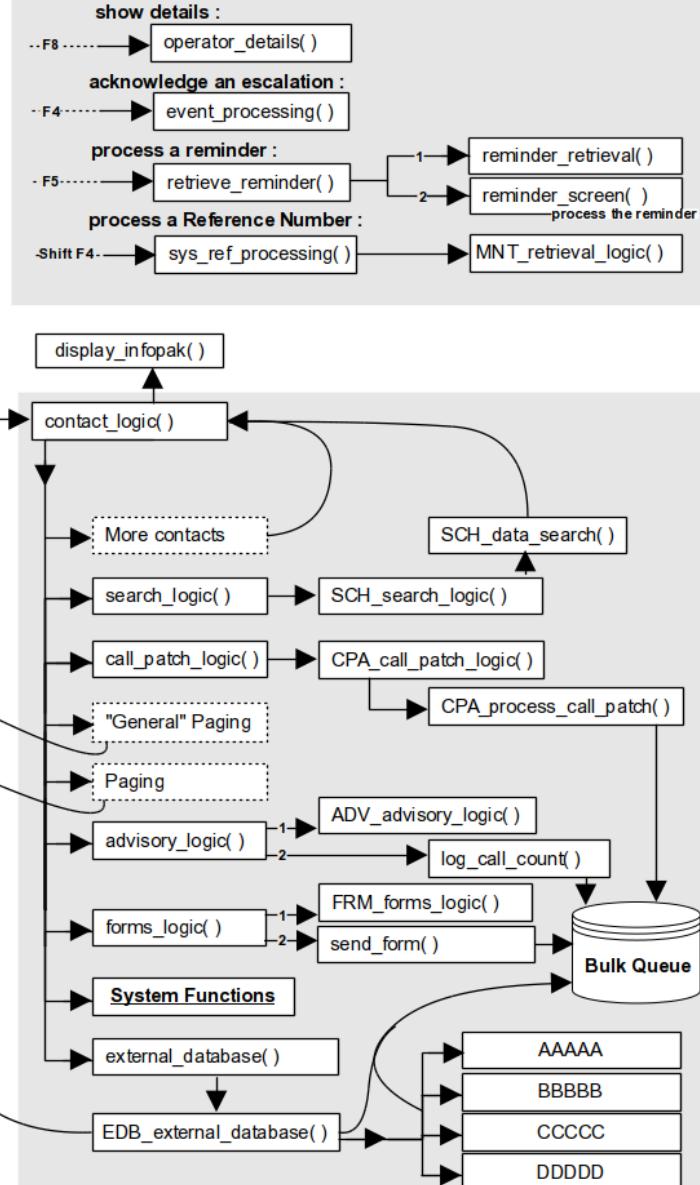
Table 17: skymds01: Key Short-cuts(1)

skymds01

Call Centre Graphical Interface to <Company 1>



SYSTEM FUNCTIONS



ONLY for personal, private use. Do not retain after use; do NOT distribute.

ONLY for use in association with hiring Stephen McGregor as a technical writer.

4.7 SIMPLE PAGING FUNCTIONS

4.7.1 FUNCTION: MANUAL_LOGIC

coded in : mds01man.c,
called by : **base_logic()**, if the user presses F2
raison d'être : prompts for and processes an indial number entered by the user / Call Centre Operator. If the indial is OK, **manual_logic()** passes control to **answer_logic()**

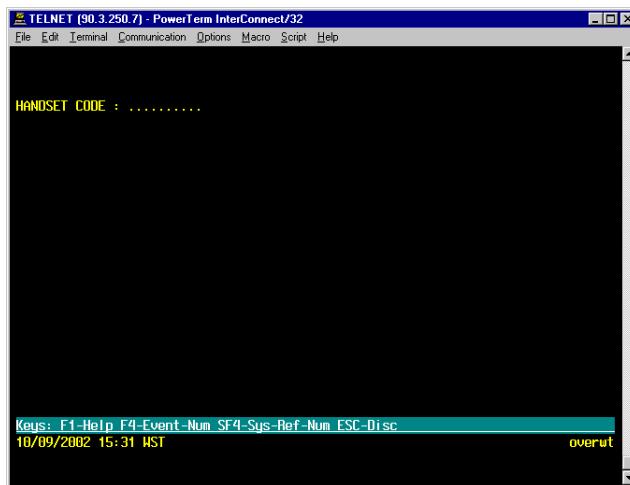
manual_logic() operates in the following manner and order:

- Set the help context so that the appropriate help for entering pager messages is displayed upon pushing F1
- start an infinite loop

control stays in **manual_logic()**, until the user pushes the ESC key enough times to escape.

- set up the input screen

do a small amount of housekeeping, clear the screen, and write the “Handset Code:” prompt for the Call Centre operator to enter the pager number.



- Read in the handset code. This is intended to be an indial (phone number), though any number appearing in the mdsline table will work.
- Either send a pager message or use the control keys to transfer to other parts of skymds01.

Key Press	Action	Description
• Control keys		
F4	acknowledge an escalation	call event_processing() . Escalation acknowledgements are “events”.
Shift-F4	process a reference number	call sys_ref_processing()
	Exit	exit manual_logic() , return to base_logic()
• Send a pager message		
9, CTRL-L	Send pager message to the previous pager-number	call answer_logic()
	read pager number and send a pager message	<ol style="list-style-type: none">1. pad the number with leading zeros2. convert some numbers to accommodate historical telephone numbers3. call man_read_details() to verify the pager number entered.4. call answer_logic() to carry on sending a pager message.

Table 18: skymds01: Key Short-cuts (2)

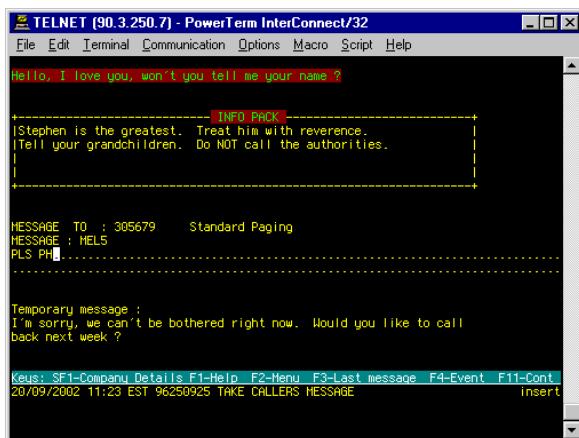
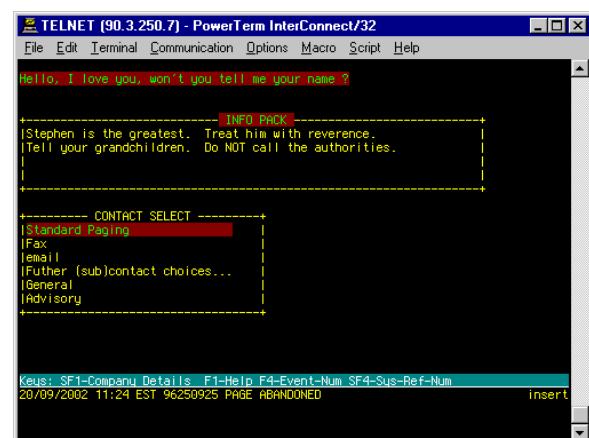
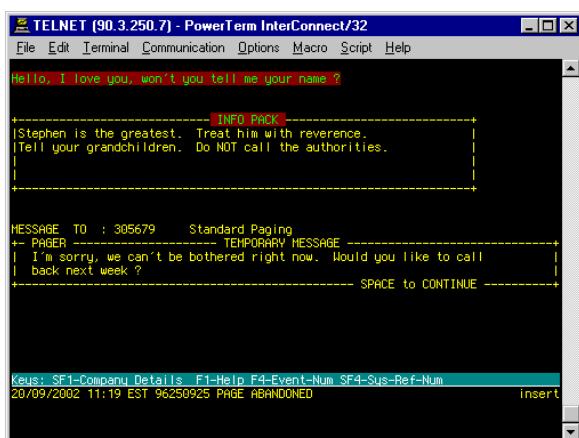
4.7.2 FUNCTION: ANSWER_LOGIC

coded in : skymds01.c file
 called by : **manual_logic()**: user presses F2 or either Ctrl-L or Shift-F9 (using the previous pager number)
 : **general_logic()** if the MDS_GENERAL_LINE environment variable does not match an indial pre-existing in the mdsline database.
 : **base_logic()** if the user presses or Ctrl-L or F3 to accept a call from the PABX
 : **general_pager_select()**, if the user has selected line mode and enters a (valid) line number
 raison d'être : **answer_logic()** contains the logic for handling an answered paging call, i.e. sending a pager message

answer_logic() is the entry point for managing the incoming telephone calls, and it is from here that much of the <System_Name> pager functionality is accessed. There are two (important) paths through **answer_logic()**:

1. Straight to general paging (**general_pager_select()**)
2. To contact (advanced message delivery services): **contact_logic()**

answer logic() looks like (well, its sub-functions look like):



Straight to Paging (**general_pager_select()**)

To contact processing (**contact_logic()**)

These are the types of screens that **answer_logic()**'s sub-functions display. Note that **answer_logic()** does not display any screens itself.

4.7.2.1 Function: answer_logic

- start the call-timer; for the Call Centre Operator, not the caller
- record the pager number for easy redial
- start an infinite loop, and test for loop termination
- **disp_answer_phrase()** generates and displays the appropriate answer phrase for this caller

A phrase, such as “Good *Afternoon*, Jim’s Plumbing, may I have your address please?” is generated by **disp_answer_phrase()** from various database fields. This phrase, plus the optional “info-pack” and ‘contact’ options all appear to the user at (what seems like) the same time. Screen-shots are in the **contact_logic()** section.

- **pager_selected_logic()** (is not) called to obtain the available pager numbers and services for this indial

The parameter ‘pager_selected’ is tested before this function is called. This variable, indicating whether the pager number has been already obtained, seems to always be false in all the calls to this function, so **pager_selected_logic()** will not, usually, be called.

- **general_pager_select()** is called for those who have ‘General Paging’ type services
- Two calls are made to **contact_logic()** for different types of services.

Many accounts have one or more ‘contacts’, advanced message delivery. This function collects and dispatches the messages to the bulk queue. More details, including screen-shots, are in the separate discussion below.

- Next messages via remote connections to <System_Name> are handled

These will have a single pager, else they will be handled by the contact processing code

- A series of checks and balances, handling and noting unusual cases
- Loop through again, though in most cases will exit back to the calling function
- Record the time and a CDR

4.7.3 FUNCTION: CONTACT_LOGIC

coded in : mds01con.c
called by : **SCH_data_search()** for some reason,
answer_logic() if the pager has any ‘contact’ services registered, or if they don’t but do have an ‘info-pack’ to display, as **contact_logic** displays the info_packs
raison d’être : displays the list of contact types available for this indial, processes the user’s choice, and passes control to the appropriate sub-routine. The “info-pack” for this indial⁵, if there is one, is also displayed.

This routine is for handling contacts, not standard, simple pager messages, though some contact types may have (simple) paging as part of their message delivery mechanism. If the mdpline database table indicates that the current indial is only registered for paging with no other services (‘contacts’), **contact_logic()** is not called by **answer_logic()**.

contact_logic() operates in the following manner:

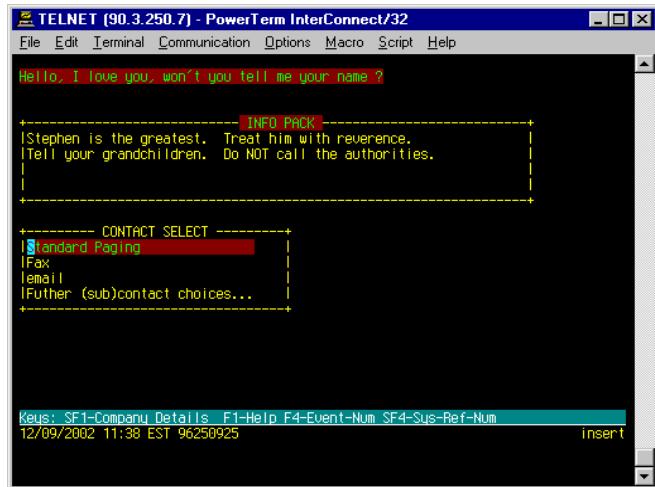
- call **display_infopack()** to display the info-pack information – if any
- list the services available (‘contacts’)

In the code, this is quite a long-winded process:

1. For people who have called in on indials with more than one service type, these services are listed.
2. there is some code for those who passed with **sub_client_flag = FALSE**, but this is an archaeological remnant.
3. the contacts are copied into a local array of contact information
4. a separate array of char pointers are set to point to each contact name string
5. the list of contacts is displayed to the user

5

An info-pack is some advice to the Call Centre Operator to help the call be processed properly. Examples might be “Most callers are elderly, be very patient” or “Seek supervisor OK before transferring to number xxxx-yyyy”



The screen-shot displays a screen containing

- A set greeting, read from the database. If no greeting was set, this text would not be displayed.
- An info-pack (info-pak), displaying information for the Call Centre Operator. If no info-pak/info-pak text was available, this box (curses ‘window’) would not be displayed.
- A contact selection curses window. If only one service was set up, this box would not be displayed, and control would transfer immediately to the message input screen

The user’s key selection is now processed, transferring control to one of 10 or so message delivery sub routines, or to one of three control functions.

- Loop starts here.

Contact: Recursive message type choices (CLT)

Present the user with another, further list of contacts. This could be used where one of the options in the original contact list was titled “Technical Support”, and, after being chosen, a list of ways to contact technical support people is presented

Paging (PAG)

Call **pager_selected_logic()**, which prompts for, collects, and sends a pager message entered by a Call Centre Operator. If the user ESC-aped out, or quit in some other way from **pager_selected_logic()**, then set flags so that this routine (**contact_logic()**) will return to waiting for the next caller’s indial. If everything went OK then write a CDR (billing) record and return to waiting for the next user choice from the “Contact Select” list.

Forms: present a customised (series of) form(s) (FRM)

Call **forms_logic()**. Like Paging, if the user ESC-aped out, or quit in some other way, **contact_logic()** returns to waiting for the next caller’s indial. If everything went OK then write a CDR (billing) record and return to waiting for the next user choice from the “Contact Select” list.

Call Centre Operator makes a call: “Call Patch” (CPA)

Call Patches are instructions for the Call Centre Operator to make a call in response to a caller’s option selection. Control is passed to **call_patch_logic()** and, like the other options, upon failure **contact_logic()** is exited while success leads to a CDR being written and the user returning to the “Contact Options” list ready to make another messaging choice.

Search (SCH)

search_logic(). Search for a pager holder’s name that starts with the text entered so far.

Read messages from an external database (VAR)

This option calls `external_database()` (in mds01edb.c), which calls `EDB_external_database()`, which, reading the mdscuprec (customer program) database table, spawns the program listed in the table record matching the indial and contact-ID collected from the user. The program spawned must write 0 or more correctly formatted bulk queue messages to a specified area of shared memory, generally by reading an external database for messages, which it (the spawned program) correctly interprets, formats and writes to this shared memory.

`EDB_external_database()` completes by calling `EDB_queue_message()` to transcribe the messages from shared memory to the <System_Name> bulk queue.

Discussion of the existing programs to read external databases is in the section on the `external_database()` function.

Advisories: Call Centre Operator reads a message to the caller (ADV)

Advisories are messages presented to the Call Centre Operator in order to be read back to the caller. When an Advisory option is chosen, `advisory_logic()` (in mds01adv.c) is called which reads the advisory message from the database and calls `ADV_advisory_logic()` to handle the task of displaying the message.

Another Standard Paging: General Paging (GEN)

`general_pager_select()` is called which, in turn, calls `pager_selected_logic()`, as does the standard paging option at the start of this list. Here one gets to enter an arbitrary pager number as a destination, c.f. the default pager number being selected from the mdsline (indial) database table.

- ... or Handle System Keys

Key Press	Action	Description
Shift-F1	Company Details	call <code>company_details()</code>
F4	acknowledge an escalation	call <code>event_processing()</code> . Escalation acknowledgements are “events”.
Shift-F4	process a reference number	call <code>sys_ref_processing()</code>
	Exit	exit manual logic(), return to base_logic()

Table 19: skymds01: Key Short-cuts (3): Control keys

- Add details of the contact message just proceeded to the current CDR record.
- Set up a dummy pager record and pass it to `pager_selected_logic()`.

`pager_selected_logic()` handles the remainder of the pager message processing, including writing the message to the bulk queue. A dummy pager message is set up because, as a contact is not a pager message⁶, it needs to be handled differently, while being passed in the same way to `pager_selected_logic()` to (uniformly) complete its passage towards the bulk queue. So it is formatted as a ‘dummy’ pager message.

- Call `pager_selected_logic()` to pass the pager message to the bulk queue.
- Store the CDR (billing) record.
- Go to the start of the loop again, unless flagged to not go through again.
- Clean up the screen, and exit

return to the calling function (`answer_logic()`), which will either re-display the “Contact Select” list, or, if there are no contacts for this indial, exit back to waiting for the next pager number to be entered.

⁶ Even if it actually is. Contacts (messages delivered via advanced handling systems) are billed differently, so the demarcation (pager c.f. contact-pager) needs to be maintained.

4.7.4 FUNCTION: PAGER_SELECTED_LOGIC

coded in : skymds01.c
called by :
: **general_pager_select()**,
: **answer_logic()** though usually the flow is:
: **answer_logic()** || **general_pager_select()** || **pager_selected_logic()**
: **single_pager_select()**. Sending pager messages from line-numbers (indials) with only one pager attached to them; thus the pager number is known.
: **contact_logic()**, when the user has chosen the standard paging option.
raison d'être : After the pager number has been obtained **pager_selected_logic()** handles all of pager message transmittal.

Once `skymds01`, the Call Centre Operator and the caller have together decided, by what-ever means, that, yes, indeed, they are going to send a pager message, `pager_selected_logic()` handles the rest of the process. In the context of `pager_selected_logic()`, ‘pager message’ includes dummy pager messages: Faxes, emails, escalations, and rostered message pretending to be pager messages. The pager number has already been collected and tested before being passed to this function, so this is purely the business end of getting the message sent.

There are two paths through this Function:

1. Go to the ‘contact’ (advanced processing) systems and set-up indicated by the code gathered from the user in `get_pager_num()`, passed to `general_pager_select()` and now here to `pager_selected_logic()`. This is pretty much a copy of `MNT_maintenance_window()`.
 2. Prompt the Call Centre Operator (or remote user) with the screen into which they should type a pager message

pager_selected_logic() operates in the following manner:

- close any unnecessary, no-longer-required or seldom-used database tables.
 - display the relevant details

‘MESSAGE:’ prompts, pager number, the destination (city and frequency – in the following screen-shot Melbourne is the city and the message will be transmitted on frequency 5)

- display the destination of the message.
 - if there is a temporary message⁷, display it.
 - if the caller can send SMS messages, connect to the SMS security shared memory block.
 - Test whether the code passed into `pager_selected_logic()` indicates that ‘contact’ processing is required:

A (fastsel) code of FAST_NONE:
Any other valid (fastsel) code:

no ‘contact’ processing. Do normal pager-message entry
yes, do alternative processing and exit.

4.7.4.1 Function: pager_selected_logic: Contact processing

For each code – each type of ‘contact’ – it is the same. Check the security level of the caller, checking twice for SMS subscribers, before calling the appropriate MNT_* function, e.g. a FAST_RETRIEVAL code (Message retrieval) will trigger the **MNT_retrieval_logic()** function, which spawns the program skymds01a to activate the message retrieval functionality. See the table below for more detail.

Task	code ('fastsel' or 'sel')	will trigger handler:	program finally called
Message Retrieval	FAST_RETRIEVAL	MNT_retrieval_logic()	skymds01a
Book a call	FAST_BOOKED	MNT_booked_maintenance()	skymds01b
Temporary Messages	FAST_SPECIAL	MNT_tmp_msg_maintenance()	skymds011
Pager (holder) name	FAST HOLDER	MNT_holder_maintenance()	skymds01d
'Follow Me': move cities	FAST_AWAY	MNT_away_maintenance()	skymds01e
Forward to another pager	FAST_TALT	MNT_talt_maintenance()	skymds01f
Send to a group of pagers	FAST GROUP	MNT_group_maintenance()	skymds01g

⁷ A Temporary Message is something like “John Brown is on holiday until Monday the 23rd. If this is an emergency...”. They are temporary (c.f. permanent) messages.

	will trigger handler:	program finally called
<u>Not called from 'pager_selected_logic()'</u>	MNT_pager_details_maintenance())	skymds01p
<u>No longer used in <System_Name></u>	MNT_canned_maintenance()	skymds01k

Table 20: skymds01: Contact codes & functions called

The same functionality appears after pushing F2 during pager message entry. See `MNT_maintenance_window()`. After the handler function (and program that the handler calls) returns, `pager_selected_logic()` exits.

4.7.4.2 Function: `pager_selected_logic` - normal pager message processing

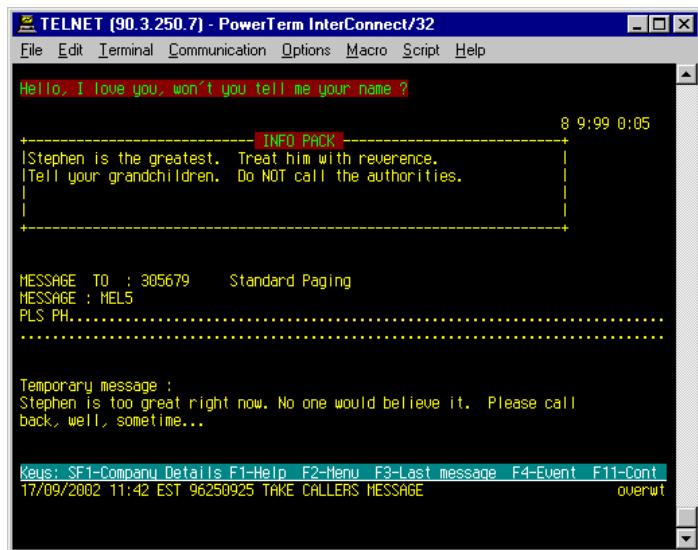
The remaining part of `pager_selected_logic()` deals with standard pager message processing

- Update the display,

different key press options are listed for remote or local users, no event or escalation processing for non-<Company_Name> phones

- optionally set a pointer to a message-retrieval function
- work out what services are available for this caller / indial
- call `get_pager_message()` to get the pager message

`get_pager_message()` (*q.v.*), coded in mdssent.c, displays the prompts and returns a pager message. If no message is entered, but return is hit, then this bit of code loops until a message is entered, or the user ESC-apes out.



- This screen-shot includes an
- info pack (optional) displaying information for the Call Centre Operator,
 - a temporary message, (optional) which is to be read out to the caller
 - The lines on which to enter the pager message, displayed by `get_pager_message`

- ESC key: exit `pager_selected_logic()`
- call `setup_queue_rec()` to prepare a bulk queue message, then `queue_main()` to post to the bulk queue.
- Use `add_client_stats()` to record that a pager message has been sent by this customer

A record of the destination pager for each pager message is made in the history and CDR when the message leaves the bulk queue.

- A couple of bits of house keeping and all done.

4.7.5 FUNCTION: GET_PAGER_NUM

Coded in : get_pager.c

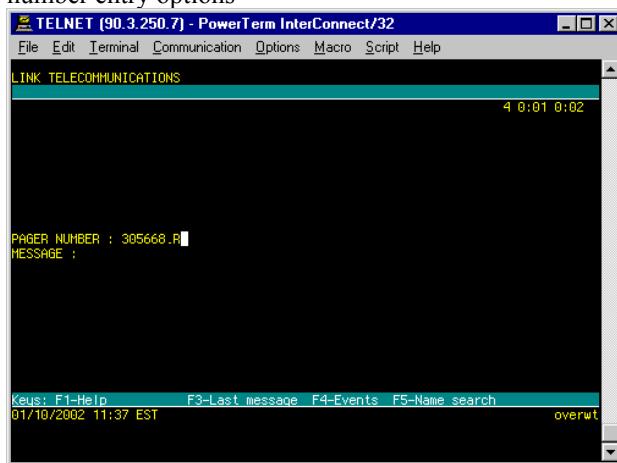
Called by : general_pager_select()

raison d'être : get – and check – a pager number from a user, plus an optional action code (a 'fastselect' code)

this function reads in and processes the pager-number entered by the user, passing back a checked pager-number, to the calling function. This action-code largely determines the behaviour of the calling function **general_pager_select()**.

Pattern	Meaning
.SOME NAME	Search for some name
SOME NAME	If in Search mode – Must have pressed F5. Otherwise all characters are errors.
12345	Send a pager message to this pager number
12345 (search)	Search for a pager <u>name</u> starting with 12345 (user has pressed F5)
12345.X	Pass a code representing ⁸ X back to the calling function, in addition to the pager number. These codes indicate services, e.g. activate the roster service (if X = "O").

Table 21: skymds01: Pager number entry options



get_pager_num() passes through the following steps:

- Display and check any number that the calling function has automatically entered
- Read and process each keystroke, one by one.

Keystrokes are of three types, numbers, control keys, and letters.

Control Keys:

ESC.	(1) erase any numbers entered so far (2) exit get_pager_num()
Del, '←'	Delete the previous character
CR (return)	Submit whatever has been entered so far
' . '	1. at the start of a name: Search on that name 2. at the end of a pager number (followed by a letter): pass a 'fastselect' code back to the calling function

Effectively a leading dot puts **get_pager_num()** into search mode, while a trailing dot will put the **general_pager_select()** function into 'Contact' mode.

Numbers: Any number is OK, except as a trailing 56789.X code, when **get_pager_num()** ignores numbers.

Letters: If after pushing F5 – indicating a search
after an initial dot '.' – indicating a search
after a trailing dot '.' – indicating a service, e.g. 'B' to Book a page.

Currently, after an initial dot '.' has been used, **get_pager_num()** stays in search mode until the user exits (ESC) and re-enters.

Table 22: skymds01: Control key-press table

- pad the pager number with leading zero, though unseen by the user

⁸ The Letter codes are translated into negative numbers (e.g. 'O' becomes -9). The calling function tests for negativity to process the various services.

ONLY for personal, private use. Do not retain after use; do NOT distribute.

ONLY for use in association with hiring Stephen McGregor as a technical writer.

4.7.6 FUNCTION: GET_PAGER_MESSAGE

coded in : messent.c
called by : **pager_selected_logic()** when not dealing with a contact – just get a single pager message.
booked_add() in skymds01b, to collect the pager message to send as a booked pager message.
add_tmsg() in skymds01l, to collect the text for the temporary message.
update_tmsg() in skymds01, to collect the updated temporary message.
raison d'être : prompts for and retrieves a message from the user / Call Centre Operator.

Usually a pager message, but any text message can be retrieved. This function is actually the skymds01 generic text entry function, supplying assistance by implementing a few short-cuts as described below. 99.9% of the calls to **get_pager_message()** are made by **pager_selected_logic()** collecting pager messages. The other two uses of this routine are to get the pager messages for booked pager calls⁹ and to collect the text for temporary messages¹⁰.

get_pager_message() operates in the following manner and order:

- Display the message-entry line and any default text (e.g. "PLS PH")
- Set the help context so that the F1 key will display appropriate help text for the current message entry

Start a loop, getting valid keystrokes and processing them

- Get a (another) valid key stroke and add it to the message being typed
- Process the keystrokes

[See the key-stroke table on the next page]

- copy the message into a static buffer for possible 'previous message' retrieval next time
- exit the function

As the text has been added to an array (char*) passed in, we simply exit and the message is available to the caller.

4.7.7 FUNCTION: GENERAL_PAGER_SELECT

coded in : skymds01.c
called by : **contact_logic()**, when implementing "general" paging
: **answer_logic()**, if the mdsline database table record for the current indial has a 'service_type' of 'G' (General Paging).
raison d'être : implements standard general paging; where the pager number is unknown and prompted for, rather than extracted from the database

general_pager_select() implements standard, simple general paging. After getting the pager number from **get_pager_num()**, it transfers to **pager_selected_logic()**. The rest is pretty much filler.

This function is seldom called, the pager number usually being available from the callers' indial (supplied by the PABX or the Call Centre Operator). It is used where

- a caller has a service set up such that they can send pager messages to one of many – or any – pagers, not just a single pager. This will usually be one of the list of 'contacts'.
- many completely independent customers are sharing the same indial.
- a real or dummy indial is set up as service_type = GENERAL_PAGING (G).

general_pager_select() operates in the following manner:

- Update the display. Accommodate for remote or local connection and display the appropriate prompts
- Get the pager number and, optionally, a 'contact' (advanced processing) code

⁹ Booked calls: Pager messages to be sent at a particular time in the future.

¹⁰ Temporary Message: A temporary "out-of-the-office" type message. No set length, but say for a few weeks while on holiday, or every Wednesday afternoon – though a Roster (*q.v.*) would probably be set up in that case.

4.7.7.1 Master (Message entry) Key-press table

Key	Function
<u>Normal Text</u>	
[A-Z,0-9] punctuation	Add the character to the message, or insert it if in ‘insert’ mode. Add the punctuation or special character to the message, or insert it if in ‘insert’ mode.
<u>Submit Messages</u>	
CR (return)	either: or (if at end of string)
	move to the end of the string exit <code>get_pager_message()</code> , returning the pager message to the calling function. The default behaviour of <code>pager_selected_logic()</code> after receiving a message from <code>get_pager_message()</code> is to exit out of processing messages for the current caller and prompt (or wait) for the next caller’s pager number.
F11	treat as CR, but clear a flag (clear_down_flag) so that <code>pager_selected_logic()</code> will continue processing messages by the current caller.
<u>Editing</u>	
Del, → , ← ↑ ↓ backspace	Normal functions
Tab	tab backwards
Ctrl-C	Clear all text entered so far.
<u>Text shortcuts</u>	
F5	add “YOU KNOW NUMBER”
Shift-F5	add “TEST MESSAGE”
F6	add “ON MOBILE”
F7	add “WHEN CONV”
F8	add “PLS PH”
F9	add “URGENT”
F10	add “RET YR CALL”
Shift-F10	add “THIS IS YOUR WAKE UP CALL FROM <COMPANY_NAME>”
Shift-F2	add some special text to the message, e.g. “MERRY CHRISTMAS”
<u>Special Functions</u>	
F2	Call a maintenance display function through a function pointer passed into <code>get_pager_message()</code>
F4	Activate Event Processing via a function pointer.
Shift-F4	Activate something (escalation handling) via a function pointer.
Shift-F1	Display Company Details.
F3	Display the previous message via a function pointer passed in.
Shift-F9	Display the previous message via local message storage.
Ctrl-L	Display the previous message via local message storage.
Ctrl-O	Toggle a display of additional lines of function-key descriptions on and off.
ESC	Clear the text and exit the function without passing a message back to the calling function.

Table 23: skymds01: Master key-press table

If the user is in ‘Line Number Entry Mode’, the `get_line()` function is used to read the pager number into the global `line_num` variable. Else, `get_pager_num()` is used to read the pager number, optionally collect a code that may indicate advanced ‘contact’ processing or just record the last key pressed (ESC, Return, F4 ...).

- Continue processing based on the contact code returned by `get_pager_num()`.

The code returned from `get_pager_num()` indicates either the key pushed, the action (Fastsel) code appended to the pager number, or a normal control key (F2, Shift-F4, etc). For standard paging, the return code is simply <CR> (i.e. return), which calls `pager_selected_logic()`. For contact processing, `pager_selected_logic()` is also called, but the Fastsel code is passed to activate the contact maintenance functions. Finally, the normal control functions are available, as codes indicating F2, F4 and the like may also be returned. See the table following.

Key	Action	Description
Fastsel code	Contact Maintenance	A specific contact maintenance function is activated depending on the code returned.
CR (Return)	In ‘Line Mode’:	call <code>answer_logic()</code>
	In Normal Mode:	call <code>check_pager_and_security()</code> , then, to continue collecting the pager message call <code>pager_selected_logic()</code> .
F2	Contact Maintenance	After successfully calling <code>check_pager_and_security()</code> , <code>associate_details()</code> is called to gather information about this pager number’s record from the database. Once the information has been gathered (into global variables) <code>MNT_maintenance_window()</code> is called to display a menu of set-up options.
F3	Get last message	via <code>display_last_message()</code> .
F4	Acknowledge escalation	call <code>event_processing()</code>
Shift-F4	Input Ref No	call <code>sys_ref_processing()</code> to input and process a reference number.
Shift-F10	Toggle Line Mode	Turns ‘Line Mode’ on, but the code to turn it off again, i.e. toggle it, does not seem to be functional.
ESC	(1) erase any numbers entered so far (2) exit general_pager_select()	

Table 24: skymds01: general paging key-press options

- return the code (last key-press or contact code) gathered by `get_pager_num()`

The function that called `general_pager_select()`, that is `answer_logic()` or the general paging section of `contact_logic()` can now process this code further.

4.8 CONTACT / ADVANCED PROCESSING FUNCTIONS

4.8.1 FUNCTION: FORMS_LOGIC

coded in : mds01frm.c
called by : contact_logic
raison d'être : Calls **FRM_forms_logic()** to presents a series of customised forms to the Call Centre Operator in order to gather and dispatch custom information.

4.8.1.1 Database Tables used

Database Table	Header File	Description
mdsfmtxt	mdsfrec.h	Holds curses (text) screens as text records
mdsforms	mdsfmrec.h	Form Details
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsqxx.txt	mdsqrec.h	Destination queues
mdsunqn	mdsunrec.h	Unique number

Table 25: skymds01 Advanced Features: form processing database tables

4.8.1.2 Function: forms_logic

- check that the mdsforms table (information about the forms) and the mdsvfmtxt (forms as collections of text records) are both open
- read the (first) form appropriate to the current indial
- if required, get a unique number (from the unique number table) to use as a system reference number.

For some types of services, callers will want to be given a reference number for their records so that when they call again they can supply their ('system') reference number to the Call Centre Operator, who, after pushing Shift F4 (system reference number processing *q.v.*) and entering the reference number gets the information supplied by the customer previously presented to them as comma delimited text¹¹. The system reference number can be passed on to the client for their records.

- check the pager and its security
- call **FRM_forms_logic()** to construct the form, present it to the user, and gather the message text
- if it is OK, call **send_form()** to post the message (derived from the data entered in the form) to the bulk queue.

4.8.1.3 Function: FRM_forms_logic

Coded in mdsfrm.c, **FRM_forms_logic()** displays the Form(s) on the screen and gathers the message(s) entered on them

- Creates the forms from the database tables
- Displays those forms
- Get the message entered by the Call Centre Operator

The message is passed back to **forms_logic()**.

4.8.1.4 Function: send_form

send_form():

- constructs a dummy pager message,
- calls **queue_main()** to post it to the bulk queue
- uses **add_client_stats()** to record a forms pager message has been sent
- optionally calls **event_processing()** to acknowledge an event number

¹¹ not embedded in the form

4.8.2 FUNCTION: SEARCH_LOGIC

coded in : mds01sch.c and mdssch.c

called by : **contact_logic()**

raison d'être : Select the appropriate destination pager given information about the pager holder.

To find the appropriate call destination when multiple people are attached to a single indial, **search_logic()** is called for the Call Centre Operator to enter details of the target as described by the caller ("She's one of the Area Managers in Victoria, and her first name is Kelly"). The search functionality allows users to search on database fields for matching records and return a pager number.

4.8.2.1 database tables used

Database Table	Header File	Description
mdsschp	mdssprec.h	Search parameter master file
mdsline	mdslirec.h	Indial number information
mdsschd	mdssdrec.h	Search Data master file

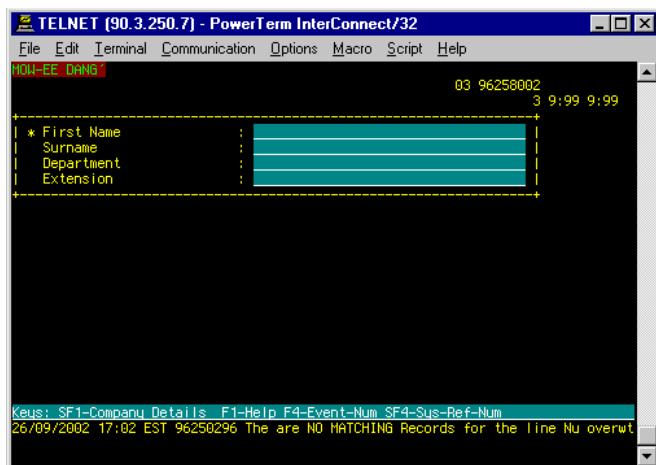
Table 26: skymds01 Advanced Features: Search database tables used

4.8.2.2 Function: search_logic

- check that the Search parameter master file (mdsschp) is open
- read the parameter record in the search parameter master file corresponding to the current indial
- call **SCH_search_logic()** to retrieve and display the data as per the search parameter record

4.8.2.3 Function: SCH_search_logic

- open mdsschd, the search data master file
- Create and display the field selection window, and fill with the appropriate prompts (e.g. "First Name", "Surname") taken from the search detail record.



- call **SCH_process_search()**, which calls **SCH_data_search()** repeatedly. **SCH_data_search()** actually does the task.

4.8.2.4 Function: SCH_data_search

SCH_data_search() looks for database fields matching one or more of the display fields, chooses the best match, and calls up the list of contacts (advanced call processing services) for the best match.

- The four search fields entered in the curses window are copied to a SDREC record in preparation for searching
- A record matching this data is looked for in the mdsschd table (search data master).
- Look around in the mdsschd table to investigate how many – if any – matching records are present
- See if there are exact matches, not just the index fields. **SCH_find_matching_rec()** does this.
- If a single exact match is found, this data is passed to **contact_logic()**.

- If there are multiple exact matches `SCH_select_search_data()` is displayed to the user to let them choose which record (person) they want. The data for the person they have chosen is written to the search screen, and `contact_logic()` called for that person
- If there are no exact matches `SCH_select_search_data()` is also called to let the user find the best person, and `contact_logic()` is called for that choice.

4.8.3 FUNCTION: CALL_PATCH_LOGIC

coded in: mds01cpa.c and mdscpa.c

called by: `contact_logic()`

raison d'être : Provide an interface to enable Call Centre Operator's to connect outside callers to another (external) line

4.8.3.1 Database tables used

Database table	Header file	Description
CDR data table	mdscdrec.h	CDR data table
mdscallp	mdscprec.h	Call patching master
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsqxx.txt	mdsqrec.h	Destination queues

Table 27: skymds01 Advanced Features: call-patch database tables used

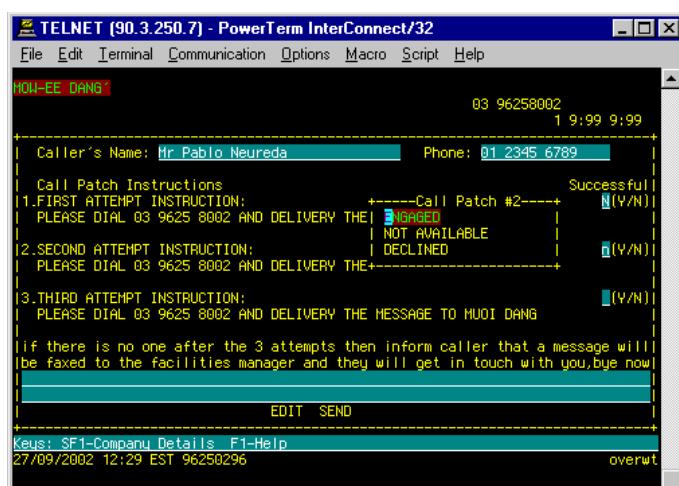
- check that the call patch file is open
- read the record for the current indial from the call patch file.
- call `CPA_call_patch_logic()` to actually implement the call patch

4.8.3.2 Function: CPA_call_patch_logic

- calls `set_global_parameters()` to collect instructions for the Call Centre Operator from the database.
- `CPA_init_call_patch_scrn()` constructs, prepares the call patch entry screen
- Finally `CPA_call_patch_logic()` call `CPA_process_call_patch()` to implement (well, manage) the call patching.

4.8.3.3 Function: CPA_process_call_patch

- `CPA_clear_screen()` clears the screen
- `CPA_display_screen()` displays the screen made by `CPA_init_call_patch_scrn()`, plus a few modifications



- The Call Centre Operator is forced to enter the name and phone number of the caller.
- `CPA_process_attempt_1()`, `CPA_process_attempt_2()`, and `CPA_process_attempt_3()` are called to collect the 1st, 2nd and 3rd attempt by the Call Centre Operator to dial the outside line. Each of these pops up a window prompting for a reason if the operator enters 'N' in the 'Successful' field.
- Upon a 'Y' answer the Call Centre Operator is transferred to the two lines at the bottom of the screen, upon which they can enter a message or note of some type.

4.8.4 FUNCTION: ADVISORY_LOGIC

coded in : mds01adv.c and mdsadv.c
 called by : **contact_logic()**
 raison d'être : Displays a message ("Advisory") for the Call Centre Operator to read to the caller and, optionally, record a brief message

4.8.4.1 Database tables used

Database table	Header file	Description
mdsadv	mdsadrec.h	Advisory
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master

Table 28: skymds01 Advanced Features: Advisory database tables used

4.8.4.2 Function: advisory_logic

- Ensure that the advisory table (mdsadv) is open
- Read the Advisory text (for this indial) from the mdsadv table
- Pass control to **ADV_advisory_logic()**
- call **log_call_count()** which pops up a text entry screen.

Call count is, apparently, Call Centre slang for a brief description of a call. In the **log_call_count()** window Call Centre Operator records a brief description of what, if anything, the caller says they were calling about. Here, if nothing is entered, the client who set up the advisory will not be billed for the call; no CDR is generated and, anyway, if the caller does not leave a message or reason for their call then we are not to know whether it was a wrong number or some type of mistake. The client could claim this anyway, so it is best to leave things as they are: no charging if no "call count" text is entered.

4.8.4.3 Function: ADV_advisory_logic

- Displays the message in a message window
- waits until the user presses ESC, or optionally process system function keys

That's it. All done.

4.8.5 FUNCTION: EXTERNAL_DATABASE

coded in : mds01edb.c (**external_database()**) and mdseedb.c (**EDB_external_database()**)
 called by : **contact_logic()**, in response to a user selecting the external database option from their contacts menu
escal_acknowledge() in skymds01a.c, Reminders, escalations and reference number processing can need the specialised handling supplied by **external_database()** functions
 raison d'être : an interface to arbitrary systems implementing customised solutions for one or more clients

The functionality contained in **EDB_external_database()** is being moved to <System_Name> II, with only four modules remaining in <System_Name>:

1. Myer Grace Bros Facilities Response Centre (FRC),
2. Safety <Company_Name>, a Personal Safety monitoring system,
3. <Company_1_Name> (cars) service request management
4. V.I.P. – A home-services franchise call handling

In the past other systems for Coca-Cola, the governmental housing body, Seleys, and others were similarly part of <System_Name>. All now moved to <System_Name> II. Some of these e.g. Myer Grace Bros, are duplicated in <System_Name> II.

external_database() extracts the current customer and line (indial) records, gets the mdscustp (customer program) record for this customer, and transfers control to **EDB_external_database()**.

EDB_external_database() operates as follows:

- connects to a shared memory block for writing the results of the custom application (success/failure)
- saves the current screen to a temporary file

It then writes the name of the temporary file to the environment variable MDS_SAVE_SCREEN, so that, if it needs to, the calling program can re-display after **EDB_external_database()** exits.

- writes the subscriber type and subscriber sub-type to environment variables
- prepares the shared memory block, writing some preliminary data
- reads the program name to run from the mdscustp (customer program) record
- calls **spawn_prog()** to run the retrieved program name
- after the external program is finished (which may be 20 minutes or more):
 1. delete the temporary file
 2. call **EDB_queue_message()** to write a group and a normal contact message to the bulk queue
 3. deletes the shared memory
- exit

4.8.6 THE FOUR (REMAINING) EXTERNAL_DATABASE PROGRAMS:

1. Myer Grace Bros Facilities Response Centre (FRC),
2. Safety <Company_Name>, a Personal Safety monitoring system,
3. <Company_1_Name> (cars) service request management
4. V.I.P. – A home-services franchise call handling

As noted, these applications are being migrated to <System_Name> II. In the past there were customised <System_Name> **external_database()** systems supporting Coca-Cola, the Commonwealth Bank, the Department of Housing, Comalco, the Environmental Protection Agency and others.

4.8.6.1 Myer Grace Bros Facilities Response Centre (FRC)

A work and resource management system developed for Myer Stores Ltd (Myer Grace Bros.). divided into two areas: “Enquiry” (How are things?) and “Maintenance” (This is how they are). The system uses customised (non-<System_Name>) logic to determine what work and resources are needed in a particular area, before using <System_Name> (including the ‘contact’ advanced handling functionality) to send out work request and status messages to the appropriate people.

Program	Source code	Functionality
<u>Used by the Call Centre Operators – called from skymds01.</u>		
cusfrcec	cusfrcec.c	Enquiry Call Processing Screen
	cusfrcmc.c	Maintenance Call Processing Screen
cusfrcwr	cusfrcwr.c	Work Request Call Processing Screen.
<u>Maintenance Screens – called from skymds00.</u>		
cusfrcfm	cusfrcfm.c	Facilities Manager Maintenance Screen.
cusfrct	cusfrct.c	Enquiry Type Maintenance Screen.
cusfrcen	cusfrcen.c	Enquiry Code Maintenance Screen.
cusfrcsi	cusfrcsi.c	Site Maintenance Screen.
cusfrces	cusfrces.c	Enquiry Code / Site Relationship Screen.
cusfrctr	cusfrctr.c	Trade Code Maintenance Screen.
cusfrema	cusfrema.c	Maintenance Code Maintenance Screen.
cusfrrsp	cusfrrsp.c	Service Provider Maintenance Screen.
cusfrcts	cusfrcts.c	Trade/Site/Service Provider Relationship Screen.
cusfrsc	cusfrsc.c	Site/Cost Centre Maintenance Screen.
cusfrcer	cusfrcer.c	Build Escalation records for Facility Response Centre

Table 29: skymds01 Advanced Features: ‘Myer’ Customised Forms functionality

Other

cusfreccg	cusfreccg.c	Cost Centre / GL Maintenance Screen.
cusfrech	cusfrccch.c	Change the Service Provider code in the Maint Log.
cusfrcfx	cusfrcfx.c	Work Request Fax

Table 30: skymds01 Advanced Features: ‘Myer’ Customised Forms functionality (other)

4.8.6.2 Safety <Company_Name> – Personal Safety monitoring systems

Safety <Company_Name> enables companies to keep track of employees, especially those working at night or in dangerous situations. The company or worker calls a <System_Name> centre with the expected schedule of a job. Once the job is completed the worker again calls a <System_Name> call centre to signal “I’m OK, job complete”. If he or she fails to call in an alarm (a “duress”) is raised and sent via <System_Name> as an escalation¹² to one or more respondents – until an “I’ve got the message” reply is received.

Program	Source code	Functionality
cussaf01	cussaf01.c	Safety <Company_Name> Login Call
cussaf02	cussaf02.c	Safety <Company_Name> Logout Call
cussaf03	cussaf03.c	Safety <Company_Name> Change/Extend Time
cussaf04	cussaf04.c	Safety <Company_Name> Change Detail
cussaf05	cussaf05.c	Safety <Company_Name> Menu
cussaf06	cussaf06.c	Safety <Company_Name> Reference Enquiry
cussafpr	cussafpr.c	Safety <Company_Name> Profile Screen
cussafsp	cussafsp.c	Safety <Company_Name> Service File Screen
cussaflg	cussaflg.c	Safety <Company_Name> log file Enquiry
cussafrp	cussafrp.c	Report Generator.

Table 31: skymds01 Advanced Features: ‘Safety <Company_Name>’ functionality

4.8.6.3 <Company_1_Name> (STARTEL)

<System_Name> handles all feedback and service request calls for <Company_1_Name> in Australia. Previously this was handled by a STARTEL system, some rare, unsupported platform with proprietary applications. Now the <Company_1_Name> functionality has now been ported to and integrated with the <System_Name> system.

The core <System_Name> functionality sends simple customer feedback messages to the <Company_1_Name> system. Service requests however, needing to locate a number of <Company_1_Name> dealerships near the caller’s reported location, as well as supporting specialised call-reporting to <Company_1_Name> head-quarters need the additional functionality provided by the programs listed in the table below.

Program	Source code	Functionality
cussta01	cussta01.c	Call Processing Screen. – called from skymds01.
cussta02	cussta02.c	X/<COMPANY_1_NAME> -Dealer Maintenance Screen – called from skymds00.
cussta04	cussta04.c	X/<COMPANY_1_NAME> - Location Maintenance Screen – called from skymds00.
cussta03	cussta03.c	fax to X or <COMPANY_1_NAME> the details of a call processed by a Call Centre Operator

Table 32: skymds01 Advanced Features: ‘<Company_1_Name>’ (‘Startel’) functionality

¹² Escalation: Successive message delivery techniques are tried until an “I’ve got the message” reply is received. Escalations ensure delivery.

ONLY for personal, private use. Do not retain after use; do NOT distribute.

ONLY for use in association with hiring Stephen McGregor as a technical writer.

4.8.6.4 V.I.P. – A home-services franchise

The fourth customised service supplied by <System_Name> is support for the VIP home services franchise system. <System_Name> processes all calls to VIP before the call details are sent on to the VIP marketing department. Separate maintenance functionality is accessible via the customer menu on the skymds00 start-up screen.

The services provided by V.I.P. franchises include:

Code	Service	Code	Service
CMC	Commercial Cleaning		Ironing
CRP	Carpet Cleaning	MCH	Mechanical repairs
EXT	Exterior House Cleaning, Footpaths etc.	MCW	Car Detailing, Vehicle Cleaning
GAR	Gardening, Lawn-mowing etc.	POL	Pool Cleaning
HSE	Interior House Cleaning	PES	Pest Control
		WIN	Window Cleaning

Table 33: skymds01 Advanced Features: ‘V.I.P.’ Service codes and the operations that are activated on each service include:

Code	Operation	Code	Operation
CAN	Cancellation	FCH	Franchise Enquiry
COM	Complaints	SEC	Second Call
EXI	Existing Customer	RFS	Rounds for Sale
QUO	New Customer		

Table 34: skymds01 Advanced Features: ‘V.I.P.’ Operation codes

The programs used are:

Program	Source code	Functionality
cusvip01	cusvip01.c	VIP Home Services Screen.
cusvip02	cusvip02.c	VIP Bulk Data Entry Screen.
cusvip03	cusvip03.c	VIP Database Maintenance Screen.
cusvip04	cusvip04.c	VIP Database Deletion.
cusvip05	cusvip05.c	VIP Suburb List.
cusvip06	cusvip06.c	VIP Database Report Initiator Screen
cusvip06a	cusvip06a.c	VIP DATABASE Report

Table 35: skymds01 Advanced Features: ‘V.I.P.’ Programs and Functionality

4.8.7 FUNCTION: MNT_MAINTENANCE_WINDOW

coded in : mdsmnt.c
 called by : [F2 generally summons the maintenance function]
 general_pager_select() upon pushing F2
 get_pager_message() via F2. Passed in as a function pointer; referenced as (*ctrlfunc)()
 verify_tone_only() via F2. Passed in as a function pointer; referenced as (*maint)()
 various functions in various cus*.c files (customised solutions)
 ADV_maintenance() (called by fill_in_advisory()), after F2 while updating Advisories
 raison d'être : manages and displays maintenance functions

MNT_maintenance_window() operates in the following manner and order:

MNT_maintenance_window() presents a list of maintenance functions for the user to choose from, in response to a F2 key-press. If F2 is pushed while entering a pager message, i.e. while in get_pager_message(), a menu of 8 or so choices are presented. However, if F2 is pressed while entering an email, Fax, reading an Advisory or executing the “Call Patch” logic (the Call Centre Operator making an outbound phone call in response to a pager message) only one option appears on the “Control” (Maintenance) menu: ‘RETRIEVE MESSAGES’. While in these functions the user can only retrieve past messages; Maintenance functionality is unavailable. There may also be some situations, when SMS messages are being sent to non-<Company_Name> telcos, that no Maintenance functionality is available. There is no hard-and-fast rule as to when this will arise.

Most of **MNT_maintenance_window()** is concerned with making up an array of Maintenance Tasks and a matching array of function pointers. Once the user has chosen from the list of strings (Maintenance Task names) the user's security for this functionality is checked before the same index value is used to select and call a function from the matching function pointer array. After the function returns (the user closes – ESCapes from – the maintenance function), the menu of maintenance options is re-presented again. The user can make multiple changes to the system before finally closing the “Control” (Maintenance) window.

This function is a copy of the contact processing in **pager_selected_logic()**.

- first, **MNT_maintenance_window()** tests whether, due to this being an SMS message en route to a non-<Company_Name> telco, Maintenance should be unavailable and **MNT_maintenance_window()** should just exit.
- next, a test is made as to whether the full range of maintenance options should be available, or just message retrieval
- Most of the remainder of the function is concerned with setting up the menu of maintenance options and pointers to the matching functions.

The protocol for maintenance function is the same. If the caller can send SMS messages, the security block (pointed to by an external Shared Memory pointer) for both SMS and normal paging are checked. If they can only send pager messages, only the paging shared memory block is checked. If these are OK, the text is added to the array of menu entries and the function pointer to the array of function pointers.

Menu Item	Function called
RETRIEVE MESSAGES	MNT_retrieval_logic()
FOLLOW ME	MNT_away_maintenance()
BOOKED PAGE	MNT_booked_maintenance()
TEMPORARY MESSAGES	MNT_tmp_msg_maintenance()
ALTERNATE PAGER	MNT_talt_maintenance()
GROUP PAGERS	MNT_group_maintenance()
CANNED MESSAGES	MNT_canned_messages()
PAGER DETAILS	MNT_pager_details_maintenance()

Table 36: skymds01 Advanced Features: maintenance menu items and functions

- The menu is displayed to the user
- The user can call one or more items (in succession) from the menu.

Once the user chooses a menu item, the corresponding Maintenance function is called. The menu stays until the user ESCapes out.

- Remove the menu and exit **MNT_maintenance_window()**

4.8.8 FUNCTIONS: **MNT_XXX_MAINTENANCE / LOGIC**

coded in :	mdssent.c		
called by :	MNT_maintenance_window(), through function pointers.		
raison d'être :	MNT_retrieval_logic() Retrieve messages MNT_away_maintenance() “Follow me” send to alternative city MNT_booked_maintenance() Set up Booked pages MNT_tmp_msg_maintenance() Maintain Temporary messages MNT_talt_maintenance() Set up alternate pagers MNT_group_maintenance() Set up group pagers MNT_pager_details_maintenance() Maintain Pager details	calls: skymds01a skymds01e skymds01b skymds01l skymds01f skymds01g skymds01p	

As indicated by the table, all of these functions are wrappers for skymds01x programs. Each does a small amount of processing, exports a number of variables to the environment before spawning a separate program to handle the maintenance functionality. The spawned programs read in the information they need from the environment variables just exported.

4.8.8.1 MNT_retrieval_logic – spawns skymds01a

- Check whether a PIN number has to be entered to retrieve previous messages

If it is, `validate_pin()` is called to prompt for and test a user-entered PIN number.

- Write a whole lot of information to the environment for skymds01a to read once it's spawned
- If the caller is an SMS customer call `display_surepage_referral()` to display a message saying something like "No can do"
- If not an SMS customer, call `run_prog()` to spawn skymds01a

4.8.8.2 MNT_away_maintenance – spawns skymds01e

- Write a whole lot of information to the environment for skymds01e to read once it's spawned
- call `run_prog()` to spawn skymds01e

No PIN or SMS checking

4.8.8.3 MNT_booked_maintenance – spawns skymds01b

- Write a whole lot of information to the environment for skymds01b to read once it's spawned
- call `run_prog()` to spawn skymds01b

No PIN or SMS checking

4.8.8.4 MNT_tmp_msg_maintenance – spawns skymds01l

- Write a whole lot of information to the environment for skymds01l to read once it's spawned
- call `run_prog()` to spawn skymds01l

No PIN or SMS checking

4.8.8.5 MNT_talt_maintenance – spawns skymds01f

- Write a whole lot of information to the environment for skymds01f to read once it's spawned
- call `run_prog()` to spawn skymds01f

No PIN or SMS checking

4.8.8.6 MNT_group_maintenance – spawns skymds01g

- Writes the pager number to the environment for skymds01g to read once it's spawned
- call `run_prog()` to spawn skymds01g

No PIN or SMS checking

4.8.8.7 MNT_pager_details_maintenance – spawns skymds01p

- Write the pager number and the State code to the environment for skymds01p to read once it's spawned
- call `run_prog()` to spawn skymds01p

No PIN or SMS checking

4.9 SYSTEM FUNCTIONS

4.9.1 FUNCTION: EVENT_PROCESSING – F4

coded in : mds01evt.c
called by : in response to an F4 key in various parts of skymds01
from:
~ contact_logic()
~ send_form()
~ manual_logic()
~ FRM_fill_in_form()
~ SCH_process_search()
~ base_logic()
~ general_pager_select()
~ pager_selected_logic()
raison d'être : to record receipt of 'escalation' responses – people calling in to acknowledge the receipt of a message.
An 'Event' is an escalation acknowledgement

Various mediums (email, pager, fax ...) send messages transferred by the Escalations system, repeatedly, until a recipient contacts a <System_Name> Call Centre acknowledging receipt. As the concept behind escalations is guaranteed message delivery, different delivery techniques may be tried, even if a Call Centre Operator or Call Centre supervisor has to start calling people personally on the phone. Thus the message will get through.

4.9.1.1 Database tables used

Database table	Header file	Description
mdspager	mdsparec.h	Pager master
mdsestrn	mdsetrec	Escalations Master File

Table 37: skymds01 Advanced Features: Escalation Acknowledgement (Events): database tables used

4.9.1.2 Function: event_processing

- Ensure that the Escalations Master database table (mdsestrn) is open
- The escalation (event) has a 7 digit number. This must be read out by the caller, recorded by the Call Centre Operator, and checked against the outstanding escalations in the escalation file.
- Prepare to retrieve the message in order to flag its acknowledgement
- retrieve it.
- all done.

4.9.2 FUNCTION: SYS_REF_PROCESSING – SHIFT F4

coded in : mds01srn.c
called by : in response to a Shift-F4 key in many functions,
viz.:
~ contact_logic()
~ manual_logic()
~ FRM_fill_in_form()
~ SCH_process_search()
~ base_logic()
~ general_pager_select()
~ pager_selected_logic()
raison d'être : to enable the Call Centre Operator to see the information entered, at some earlier date, on a customised 'form' (see **forms_logic()**). The customer was given the ("system") reference number at that time.

4.9.2.1 Database table used

Database table	Header file	Description
mdspager	mdsparec.h	Pager master
mdsestrn	mdsetrec	Escalations Master File

Table 38: skymds01 Advanced Features: Reference Numbers: Database tables used

4.9.2.2 Function: sys_ref_processing

- Present a box into which the Call Centre Operator can enter the caller's System Reference Number
- Check that the system reference number appears in the escalations database table (mdsestrn), where it is stored
- Do some preparation for retrieving the message generated
- use **MNT_retrieval_logic()** to retrieve the message from the history server.

MNT_retrieval_logic() spawns skymds01a to retrieves previous messages for an indial. In this case, and with the supplied system reference number, skymds01a retrieves the message(s) generated previously by the appropriate form. (see **forms_logic()**)

4.9.3 FUNCTION: REMINDER_RETRIEVAL – F5

coded in : mds01rem.c and mdsrem
 called by : in response to a F5 key press during the **base_logic()** function (the blank screen before pushing F2 to enter a pager number) if (and only if) one's account is set up to process reminders.
 raison d'être : supply the next 'reminder' to a Call Centre Operator for processing.

Reminders are tasks for Call Centre Operator to process, often calling a customer or contacting a supervisor with a message.

reminder retrieval():

- checks to see tminder database table is open
- calls **REM_get_next_reminder()** to get the first reminder
- uses **reminder_screen()** to display the rehat the reminder screen.

4.9.3.1 database tables used

Database table	Header file	Description
hiccyymmdd	mdshirec.h	the 'logical' history
hsyymmdd	mdshsrec.h	History secondary storage
mdscustp	mdscurec.h	Customer program
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsrem	mdsrrec.h	Reminders queue
mdsremp	mdsrprec.h	Reminders parameter

Table 39: skymds01 Advanced Features: Reminders: Database tables used

4.9.3.2 Function: REM_get_next_reminder

- Lock the reminder queue

If locking the queue fails – hopefully only due to the queue already being locked by someone else - **REM_get_next_reminder()** exits, passing a message back to **reminder_retrieval()** to prompt the user to try again.

- Find the first appropriate reminder
- Flag that it is being processed
- decrement the count for this reminder queue
- unlock the reminder queue and return

4.9.3.3 Function: reminder_screen

- ensure that the customer program table is open

The advanced handling (“PROCESS”) option for any program requires this.

-
- read the reminder
- get the CDR (billing) information
- Set up the text window. `rev_select()` is then used to display the window with four options from which the Call Centre Operator can record what happened during reminder processing:
 - ACTIONED
 - ENGAGED
 - NOT AVAIL
 - RETRIEVAL
 - PROCESS
- If the user chose the option “RETRIEVAL”, call `MNT_retrieval_logic()` to retrieve the message.

This message will probably be an instruction to a Call Centre Operator to do something, like contact their supervisor.

- If the user chose the option “PROCESS”, `edb_external_database()` is used to implement customised handling of the reminder
- For some types of reminders call `write_hist()` to write a history record.
- delete the reminder just processed.

4.10 OTHER FUNCTIONS

4.10.1 CATCH_SIGUSR1 (AND CATCH_SIGUSR1_CALLER_DROP)

Each time a call arrives on the PABX, the user defined signal SIGUSR1 is raised on the <System_Name> server. If everything is OK, the `catch_sigusr1()` signal handler is called, which passes control to the ‘stdscr’ curses window. If there is a problem, and this signal is received while the Call Centre operator is still on a call, or an operator pushes a key while waiting for a call to arrive, the `catch_sigusr1_caller_drop()` handler will be called and an error window displayed. However, with PABX updates, it is less and less likely that such a scenario will be encountered, or at least cause a problem.

4.10.2 SETUP_SIGNAL_CATCHING, CATCH_SIGNAL, CATCH_ALARM, CATCH_TERM, CATCH ...

- `setup_signal_catching()` <Company_Name>s most signals to the `catch_signal()` handler, which writes a log message and exits.
- `catch_alarm()` presents the standard ‘stdscr’ curses window, in effect acting as a wakeup call to a Call Centre operator’s skymds01 application.
- `catch_term()` deflects any further termination signals and calls cleanup to exit the program gracefully

4.10.3 CHECK_SECURITY_LEVEL

`check_security_level()`, coded in security.c takes a shared memory pointer and an index (usually as a `#define`d constant) and tests what value is in the shared memory at the location ‘pointer + index’. It then grants access (i.e. returns true) if the value at that location is ‘1’.

4.10.4 POP_HELP, PUSH_HELP, SET_HELP

The ‘help context’ indicates to the skymds01 help function what type of help to display, which obviously changes as the user interacts with different parts of the program. These functions manage the changing of the help context.

pop_help() – return to the previous help context

push_help() – set a new help context to the stack of help contexts

set_help() – set the help context to a particular value

4.10.5 DISPLAY_INFOPACK

The info-pack for an indial is a set of background information for the Call Centre Operator.

“Be very clear about the phone number. Repeat it back to the caller” or

“Offer to transfer the call to a Hospital. Do NOT give ANY advice”

are the types of information that may be displayed in an Info-pack display.

display_infopack() reads text from the appropriate record in the mdsinpak database table and writes it to the screen in a curses window.

5 skymds01a to skymds01p

The functionalities of the programs skymds01a to skymds01p are as follows:

Program	Functionality	Spawned by
skymds01a	Retrieve messages	<code>MNT_retrieval_logic()</code>
skymds01b	Set up Booked pages	<code>MNT_booked_maintenance()</code>
skymds01d	Pager Holder Name Maintenance	- not called by <code>MNT_*</code> functions -
skymds01e	“Follow me” send to alternative city	<code>MNT_away_maintenance()</code>
skymds01f	Set up alternate pagers	<code>MNT_talt_maintenance()</code>
skymds01g	Set up group pagers	<code>MNT_group_maintenance()</code>
skymds01l	Maintain Temporary messages	<code>MNT_tmp_msg_maintenance()</code>
skymds01p	Maintain Pager details	<code>MNT_pager_details_maintenance()</code>

Table 40: Functionality of skymds01? programs

As described in the skymds01 section, each of these is spawned by the corresponding `MNT_xxx_maintenance` function, and implements the management of a particular advanced service.

A diagram showing all these maintenance applications, their functions and their relationships appears at the end of this section

5.1 PROGRAM: SKYMD01A – MESSAGE RETRIEVAL

skymds01a manages message retrieval.

5.1.1 SKYMD01A: DATABASE TABLE USED

Database table	Header file	Description
mdsaparty	mdsaprec.h	A-Party customer info
mdsacrec	mdscdrec.h	Alt call details master
mdscdrec	mdscdrec.h	Call details master
mdsclists	mdscsrec.h	Client stats master
mdscustp	mdscurec.h	Customer program
mdsdestm	mdsderec.h	Destinations master
mdsestrn	mdsetrec.h	Escalation transaction master
mdsforms	mdsfmrec.h	Forms
mdsfmtxt	mdsfmrec.h	Forms text
hiccyymmdd	mdshirec.h	the ‘logical’ history
hsyymmdd	mdshsrec.h	History secondary storage
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsphone	mdsphec.h	Phone master
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsquer.c	Destination queue s
mdsrem	mdsrmrec.h	Reminders queue

Table 41: skymds01a: Message Retrieval: Database tables used

5.1.2 FUNCTION: (SKYMD01A) MAIN

Following a similar pattern to most other <System_Name> programs, skymds01a’s `main()` Function:

- calls an `initialise()` function,
- `msg_retrieval()` to execute the central logic of the program.
- Finally a `cleanup()` function is called to complete execution.

5.1.3 FUNCTION: (SKYMDS01A) INITIALISE

skymds01a's initialise function follows the same process as most other <System_Name> initialise functions.

- catch most common signals
- read in a number of environment variables, including those especially written by `MNT_retrieval_logic()`. Also read in the name of the file holding the saved previous screen image
- open all necessary database tables, opening a second index on some
- read the destination codes from the destination table: mdsdestm
- Do a special validation for Telstra
- Open the bulk queue and prepare for writing to it
- Attach to the reminders shared memory block
- Initialise cursors
- First re-establish the saved previous screen image. Now re-draw the main window, on top of which then superimpose the retrieval window

5.1.4 FUNCTION: (SKYMDS01A) MSG_RETRIEVAL

`msg_retrieval()` prints out a list of message retrieval options, and then calls appropriate functions based on the choice that the user makes.

- Construct the menu entries from strings (like "RESEND", PREV", etc)
- Two versions are made: for Surepage and for everyone else
- prepare a record to enter into the retrieval audit trail
- display the previous message for this indial
- call `rev_select()`, to see what option from the menu the user has chosen:

Menu Item	Function Called	Behaviour
NEXT	<code>msg_prev()</code>	Retrieve the previous message. <code>NHIST_read_prim()</code> is used to read the message from the history file
PREV	<code>msg_next()</code>	Retrieve the next message if we are already down the message list a bit. <code>NHIST_read_prim()</code> is used to read the message from the history file
RESEND	<code>send_msg_cmd()</code>	Resend a selected, retrieved message. Call <code>setup_queue_rec()</code> and <code>queue_main()</code> to resend the message
ACK	<code>escal_acknowledge()</code>	The Call Centre Operator uses this to record a caller's acknowledgement of having received an escalated message
AUDITS	<code>get_and_display_details_cmd()</code>	With a few other things this function repeatedly calls <code>display_details_cmd()</code> to list any and all audit records for this indial
ORIG	<code>orig_processing()</code>	saves the current program state before (recursively) calling <code>msg_retrieval()</code>
MSG SCRL	If the message is longer than the displayed field then move it along one line.	

Table 42: skymds01a: Message Retrieval Key-press Functionality

5.2 PROGRAM: SKYMDS01B – BOOKED PAGER MESSAGE MAINTENANCE

skymds01b manages booked pager messages. Having received the valid line record of a pager, it allows the user (Call Centre Operator) to add, delete, change and view booked pager messages. The booked message(s) is (are) not written to the bulk queue, instead being written to the timed events table from which they are written to the bulk queue at appropriate times.

5.2.1 DATABASE TABLES USED

Database table	Header file	Description
mdsacrec	mdscdrec.h	Alternative call details master
mdsaudit	mdsaurec.h	Audit trail
mdscdrec	mdscdrec.h	Call details master
mdspager	mdsparec.h	Pager master
mdspassw	mdspssrec.h	Password username table
mdsports	mdsporec.h	Ports security
mdstimev	mdsterec.h	Timed events

Table 43: skymds01b: Booked Pages: Database tables used

5.2.2 FUNCTION: (SKYMDS01B) MAIN

Following a similar pattern to most other <System_Name> programs, skymds01b's main() Function:

- calls an `initialise()` function,
 - calls `CDR_start_session()` to ensure that today's CDR file is open.
 - `booked_maint()` to execute the central logic of the program; managing booked pager messages
 - `CDR_end_session()` closes the CDR file
 - Finally a `cleanup()` function is called to complete execution.

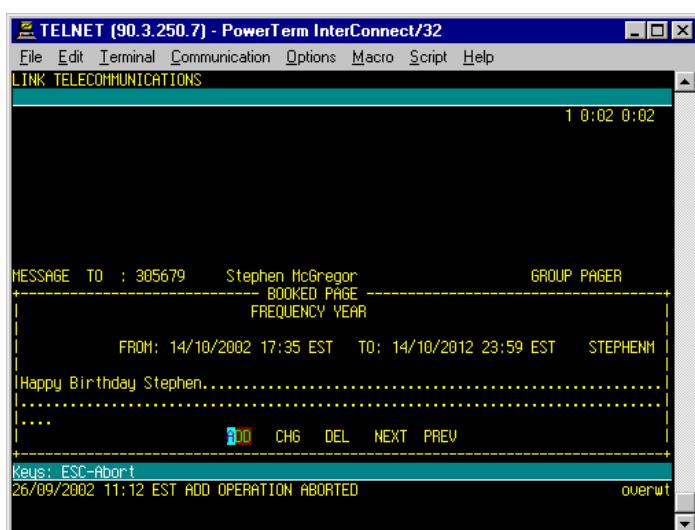
5.2.3 FUNCTION: (SKYMDS01B) INITIALISE

- catch a few signals
 - read a few environment variables, including the name of the file holding the saved previous screen image.
 - initialise the curses system with this previous screen image.
 - process the single ‘Insert/overwrite’ command-line parameter.
 - read in 18 more environment variables
 - open the timed events, pager and audit database tables
 - open the maintenance queue for sending updates between cities
 - call `get security shared mem()` to attach to the security shared memory block

5.2.4 FUNCTION: (SKYMDS01B) BOOKED MAINTENANCE

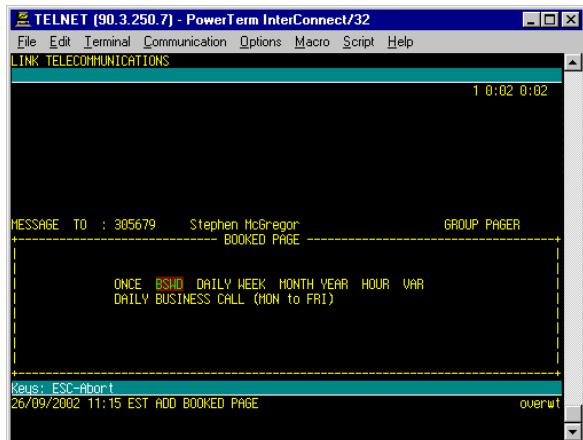
booked maintenance() provide a window and functionality for a Call Centre Operator to manage booked pager calls.

- writes the function key prompts onto the screen (“SELECT FUNCTION”, Keys: ESC-Abort”)
 - display the previous booked pager call



- calls `rev_select()` to see what menu choice (or ESC) the user has made
- The responses to the menu options are listed below
- call `display_booked_entry()` to display the booked pager message

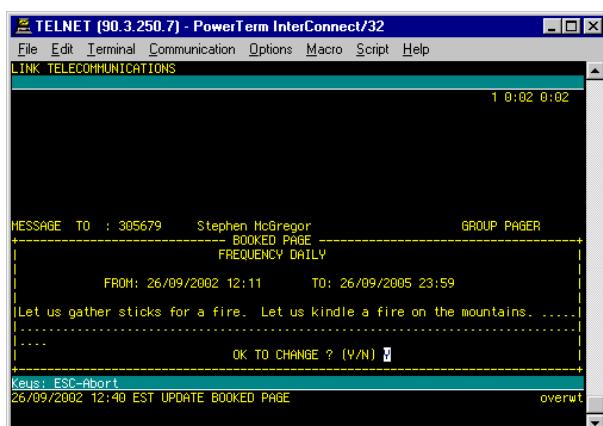
5.2.4.1 Menu Item “ADD”: calls `booked_add`



- A couple of bits of house keeping
- Sets the value “field” to 0, and then enters a switch statement selected by this “field” variable

Thus case 0 will always be called. Each case statement updates the value of “field”.

- (case 0): Print out the menu options and collect the Call Centre Operator’s selection
- (case 1): Print out the “FROM:” prompt and collect the date input
- (case 2): If the selection is for a ONCE only booked page, bail out (set field = 6), else write “TO:” collect and process the date. If the VAR (variable times) choice was made from the list of options “field” is set to 5 (enter variable minutes). Usually “field” is set to 6
- (case 6): Call `get_relevant_tzone()` to print “ARE THE TIMES RELEVANT TO VICTORIA TIME ZONE? (Y/N) Y” and collect the response. “filed” is set to 3
- (case 3): Call `get_pager_message()` to prompt the user to enter a pager message
- (case 4): Prompt “OK TO ADD? (Y/N) Y”. If the user answers ‘Y’,



- Now set up a timed events record containing the information just entered
- Write an audit record
- Write a CDR record

5.2.4.2 Menu Item “CHG”: Function booked_update

Pretty much a repeat of `booked_add()` q.v.

5.2.4.3 Menu Item “DEL”: Function booked_delete

- calls `answer_yn()` to prompt “OK TO DELETE? Y”
- if the user answers T, the C-ISAM function `isdelcurr()` delete the current booked page
- An audit record is constructed recording the deletion, and written with `write_audit()`
- a CDR record is written.

5.2.4.4 Menu Item NEXT: Function booked_next

- If no current record (i.e. just started) get to the first dated Booked pager for the current pager
- else get to the next most recent record
- check that the booked page is in fact for this number

5.2.4.5 Menu Item PREV: Function booked_prev

This option only appears if we are not on the (default) earliest booked page. It is very similar to `booked_next()`.

5.3 PROGRAM SKYMDSD01D – PAGER HOLDER NAME MAINTENANCE

skymds01d pops-up a simple window in which the user can adjust the name of the pager holder.

5.3.1 DATABASE TABLES USED

Database table	Header file	description
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdsaudit	mdsaurec.h	Audit trail

Table 44: skymds01d: Pager Holder Name Maintenance: Database tables used

5.3.2 FUNCTION: (SKYMDSD01D) MAIN

A standard <System_Name> `main()` function, with control passed to `holder_maint()` to implement the logic of the program.

- call `initialise()` to set up skymds01d’s program environment
- call `CDR_start_session()` to open the CDR file
- pass control to `holder_maint()`
- call `CDR_end_session()` to close the CDR file

5.3.3 FUNCTION: (SKYMDSD01D) HOLDER_MAINT

`holder_maint()` just displays the current name on a single line, and gives a single choice: to change it. If the user does choose to change, `holder_update()` implements the change.

5.3.4 FUNCTION: (**SKYMDS01D**) **HOLDER_UPDATE**

holder_update() uses a switch statement, with the switch value ('field') being changed after

- straight into the first section of a switch statement, which reads the alterations the user has entered on the text line
 - Prompt "OK to Change? (Y/N)"
 - N: call **display_holder_entry()** to redisplay the original name
 - ESC: return to **holder_maint()** with the original name displayed
 - Y: write a CDR
 - Update the pager record
 - write the change to the maintenance queue for transmittal to the other <System_Name> sites
 - write an audit record
-

5.4 PROGRAM SKYMDS01E – AWAY / FOLLOW ME MAINTENANCE

skymds01e manages temporary forwarding of pager messages to alternative geographical regions¹³, functionality known in <System_Name> as "Follow Me". If a pager holder wants to receive messages in a different part of Australia they must organise a Follow Me service with <Company_Name> Communication Corporation.

5.4.1 DATABASE TABLES USED

Database table	Header File	Description
mdsacrec	mdscdrec.h	Alternative call details master
mdsaparty	mdsaprec.h	A-Party Customer Info
mdsaudit	mdsaurec.h	Audit trail
mdscdrec	mdscdrec.h	Call details master
mdsdestm	mdsderec.h	Destinations master
mdsfacil	mdsfarec.h	Facilities – what special <System_Name> services can be used
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsquarec.h	Destination queues
mdstimev	mdsterec.h	Timed events

Table 45: skymds01e: Follow Me Maintenance: Database tables used

5.4.2 FUNCTION: (**SKYMDS01E**) **MAIN**

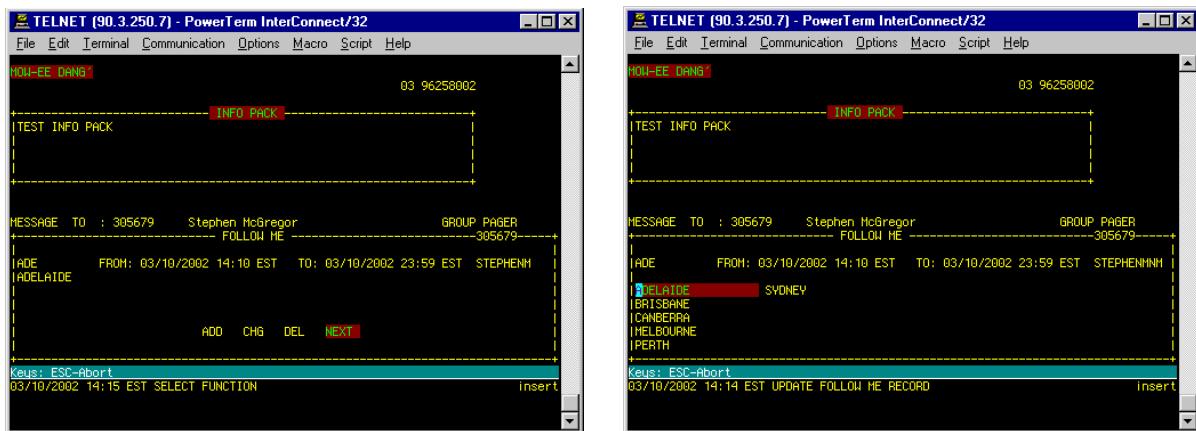
- call **initialise()** to establish skymds01e's program environment
- call **CDR_start_session()** to open the CDR file. CDRs are billing records written by many <System_Name> processes
- pass control to **away_maint()** to exercise the main logic of the program
- call **CDR_end_session()** to close the CDR file.

¹³ Pager message transmission is local and geographically dependent. A message transmitted in Melbourne will not be received in either Geelong or Brisbane.

5.4.3 FUNCTION: (SKYMDS01E) INITIALISE

`initialise()` follows the same pattern as most other <System_Name> initialise functions.

- read in a whole lot of environment variables, many written by the `MNT_away_maintenance()` function
- initialise the curses system
- parse the single “overwrite / insert” command line parameter
- read in a few more environment variables
- open all database tables needed
- open the maintenance queue for transmitting updates to other <System_Name> systems
- attach to the security shared memory segment



5.4.4 FUNCTION: (SKYMDS01E) AWAY_MAINT

`away_maint()` calls one of many functions, dependent on the choice that the user makes.

- call `read_pager_and_dests()` to read in the destination (mdsdestm) and pager (mdspager) database tables
- use `away_next()` to return the most recent ‘away’ pager forwarding record, and `display_away_entry()` to display it.

`away_next()` returns a variable indicating how many “Follow Me” (‘away’) services are set up for this pager

- prompt the user with a menu of choices: ADD, CHG, DEL, NEXT, PREV

Choice ADD:	call <code>away_add()</code> to add a Follow Me forwarding record
Choice CHG:	call <code>away_update()</code> to update a Follow Me forwarding record
Choice DEL:	call <code>away_delete()</code> to delete a Follow Me forwarding record
Choice NEXT:	call <code>away_next()</code> and then <code>display_away_entry()</code>
Choice PREV:	call <code>away_prev()</code> and then <code>display_away_entry()</code>

In the code these choices are coded twice, with and without the PREV choice.

- finally, handle the case with no ‘away’ records already on file, and refresh the tables from the database

5.5 PROGRAM SKYMDS01F – TEMPORARY ALTERNATIVE PAGER

Sometimes, for various reasons, a customer will want to forward messages, destined originally for their pager, onwards to another pager. The Temporary Alternative Pager Maintenance functionality manages this forwarding.

5.5.1 DATABASE TABLES USED

Database Table	Header File	Description
mdsacrec	mdscdrec.h	Alt call details master
mdsaudit	mdsaurec.h	Audit trail
mdscdrec	mdscdrec.h	Call details master
mdsfacil	mdsfarec.h	Facilities – what special <System_Name> services can be used
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security

Table 46: skymds01f: Temporary Alternative Pager: Database tables used

5.5.2 FUNCTION: (SKYMDS01F) MAIN

- call **initialise()** to establish skymds01f’s program environment
- call **CDR_start_session()** to open the CDR file. CDRs are billing records written by many <System_Name> processes
- pass control to **alt_main()** to exercise the main logic of the program
- call **CDR_end_session()** to close the CDR file.

5.5.3 FUNCTION: (SKYMDS01F) INITIALISE

initialise() follows the same pattern as most other <System_Name> initialise functions.

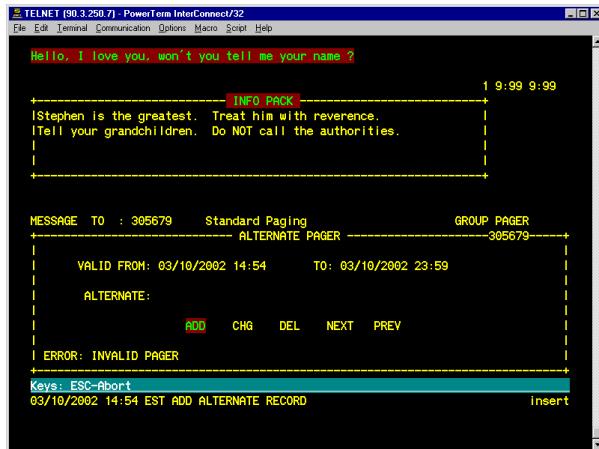
- read in a whole lot of environment variables, many written by the MNT_away_maintenance() function
- initialise the curses system
- parse the single “overwrite / insert” command line parameter
- read in a few more environment variables
- open all database tables needed
- open the maintenance queue for transmitting updates to other <System_Name> systems
- attach to the security shared memory segment

5.5.4 FUNCTION: (SKYMDS01F) ALT_MAINT

alt_main() implements the central logic of the skymds01f Temporary Pager Maintenance

- get the home city of the current pager holder from the pager (mdspager) table
- display the list of option choices to the user

Choice ADD: call **alt_add()** to add an alternative pager record
Choice CHG: call **alt_update()** to update an alternative pager record
Choice DEL: call **alt_delete()** to delete an alternative pager record
Choice NEXT: call **alt_next()** and then **display_alt_entry()** to display the next alternative pager set up.
Choice PREV: call **alt_prev()** and then **display_alt_entry()** to display the previous alternative pager set up.



- finally `alt_maint()` re-reads the database table to stay consistent with any changes that have been made

The functions called from `alt_maint()` all have pretty obvious functionality, nothing beyond what can be seen from their names and looking at the temporary maintenance screens. Just note that one can not forward to a pager that itself has an alternative pager set up.

5.6 PROGRAM: SKYMDS01G – GROUP PAGING MAINTENANCE

One type of ‘contact’ advanced processing service is the ability to send single pager messages to many pagers at once: Group Paging. The pager number becomes a reference to a group of pagers. The Group Paging Maintenance routines in skymds01g set-up and control these services.

5.6.1 DATABASE TABLES USED

Database Table	Header File	Description
mdfsfacil	mdfsarec.h	Facilities – what special <System_Name> services can be used
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security

Table 47: skymds01g: Group Paging Maintenance: Database tables used

5.6.2 FUNCTION: (SKYMDS01G) MAIN

Simple:

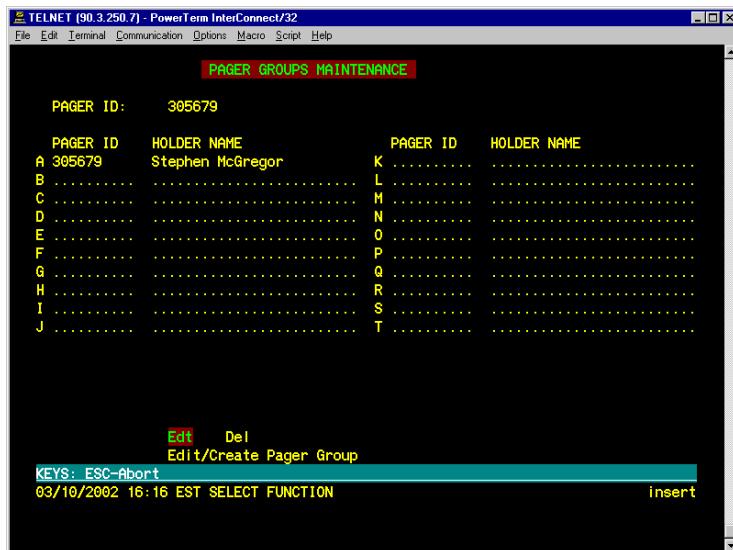
- call `initialise()`, and
- `pgroup_maint()`.

5.6.3 FUNCTION: (SKYMDS01G) INITIALISE

- Catch a few signals()
- read a few environment variables
- initialise the curses system
- update the display
- parse the single “insert / overwrite” command-line parameter
- get 10 or so more environment variables
- open all the permanently open database tables
- open the maintenance queue for replication of changes to all the <System_Name> servers

5.6.4 FUNCTION: (SKYMD01G) PGROUP_MAINT

- `pgroup_screen()` displays the basic outline of the groups maintenance screen, the title, PAGER ID, and the like
- `pgroup_read()` reads all the group members attached to the current pager number, and `display_pgroup_entry()` is used to display these group members
- `rev_select()` is now used to display the remainder of the screen, complete with a two option menu waiting for the Call Centre Operator's choice.



- Choice EDT: After using `check_security_level()`, either (for new Groups) `pgroup_add()`, which calls `pgroup_edit()`, or `pgroup_update()` (to add to an existing group) are used.
- Choice DEL: `pgroup_delete()` and `clear_pgroup_entry()` are used to remove the pager number from the group in the database and from the Call Centre Operator's screen respectively.
- finally `update_primary_pager()` is used to update the pager record with the number of pagers in the group that it represents

5.7 PROGRAM SKYMD01L – TEMPORARY MESSAGE MAINTENANCE

Temporary messages are for Out-of-Office type of requirements “John Smith is on Holiday till the 22nd. If this is an emergency” and are to be read out to the caller. In this they differ from Infopacks, which are background information only for the Call Centre Operators, and from Advisories, which are permanent messages with data collection facilities: “Please report the number of units sold and the area code”. Note that while temporary messages are time stamped with a start and end date, only one temporary message per pager can be in <System_Name> at any one time.

5.7.1 DATABASE TABLES USED

Database Table	Header file	Description
mdsacrec	mdscdrec.h	Alt call details master
mdsaudit	mdsaurec.h	Audit trail
mdscdrec	mdscdrec.h	Call details master
mdspager	mdsparec.h	Pager master
mdspassw	mdpsrec.h	Password username
mdspports	mdsporec.h	Ports security
mdstmesg	mdstmrec.h	Temporary message master

Table 48: skymds01l: Temporary Messages: Database tables used

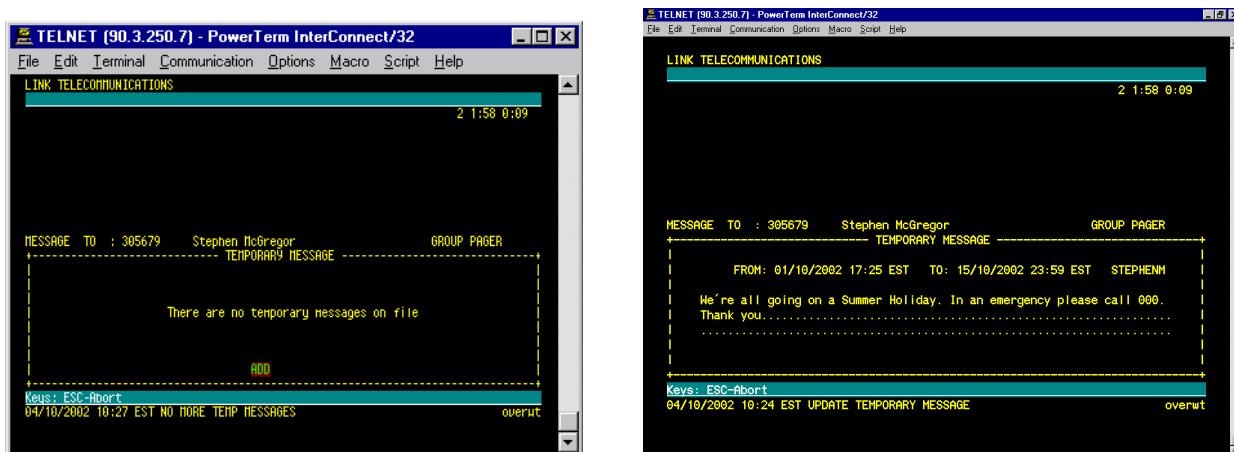
5.7.2 FUNCTION: (SKYMDS01L) MAIN

skymds01l's `main()` follows the same steps as the other skymds01x maintenance programs' `main()` functions:

- call `initialise()` to establish skymds01l's program environment
- call `CDR_start_session()` to open the CDR file. CDRs are billing records written by many <System_Name> processes
- pass control to `process_tmmsg()` to exercise the main logic of the program
- call `CDR_end_session()` to close the CDR file.

5.7.3 FUNCTION: (SKYMDS01L) INITIALISE

- read in a whole lot of environment variables, many written by the `MNT_tmmsg_maintenance()` function
- initialise the curses system and display the temporary message maintenance window
- parse the single "overwrite / insert" command line parameter
- read in a few more environment variables
- open all database tables needed
- open the maintenance queue for transmitting updates to other <System_Name> systems
- attach to the security shared memory segment



5.7.4 FUNCTION: (SKYMDS01L) PROCESS_TMSG

- call `read_tmmsg_rec()` and `read_pager_rec()` to get the temporary message record and pager record for this pager
- if there are no temporary messages on file, the user is told this and a flag is set so that they can only add temporary messages
- alternatively, if there is a temporary message, the message is displayed and the user is given the option to either change or delete the message. As noted above, one can only have one temporary message attached to a pager. Alternative temporary messages, even during non-overlapping time periods, can not be set up.
- the menu is now displayed and the users choice collected

Choice ADD: call `add_tmmsg()`

Choice CHG: call `update_tmmsg()`

Choice DEL: call `delete_tmmsg()`

Choice ESC: set a flag to leave `process_tmmsg()`, and thus leave skymds01l.

- finally `update_pager_rec()` is used to send information about any changes to the other <System_Name> servers via the maintenance queue.

5.7.5 FUNCTION: (SKYMD01L) ADD_TMSG AND update_tmsg

These two functions are practically identical.

- Display “ADD [UPDATE] TEMPORARY MESSAGE”
 - The variable “field” is set to 0, and a switch statement entered using field as its selection variable
1. FROM: date the FROM Prompt is displayed and `mnt_get_date_time()` is used to collect and check the date
 2. TO: date the TO prompt is displayed, a default date of the FROM date is displayed, before `mnt_get_date_time()` is again used to check the date (if the user has altered it).
 3. `get_pager_message()` is used as the text entry function for the temporary message, both UPDATE and ADD using the ‘message’ variable – empty in the case of ADD – as the default text.
- the temporary message for this pager is now updated
 - the change is written to the maintenance queue so that all the other <System_Name> sites are updated.

5.7.6 FUNCTION: (SKYMD01L) DELETE_TMESG

- display the prompt “OK TO DELETE? (Y/N) Y”
 - (If the answer is yes), use `mnt_queue_write()` to send the update (i.e. deletion) to the other <System_Name> servers
 - write an audit note, and log a CDR record
-

5.8 PROGRAM: SKYMD01P – PAGER DETAILS MAINTENANCE

skymds01p displays a selection of pager details, collected from various database tables, but does not provide functionality to make any changes.

5.8.1 DATABASE TABLES USED

Database Table	Header File	Description
mdsemail	mdsemrec.h	e-mail master
mdsexco	mdsecrec.h	External contact master
mdsfacil	mdsfarec.h	Facilities – what special <System_Name> services can be used
mdsfax	mdsfxrec.h	Fax master
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security

Table 49: skymds01f: Pager Details: Database tables used

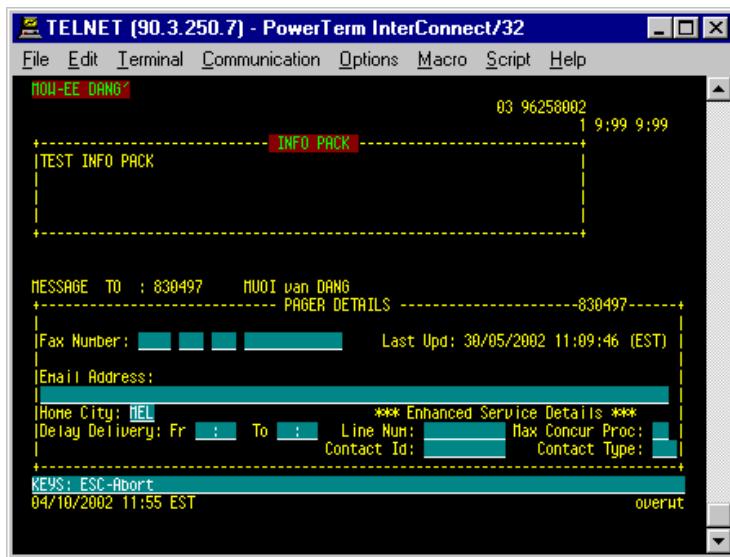
5.8.2 FUNCTION: (SKYMD01P) MAIN

`main()` only calls its `initialise()` function and then calls `display_pager_details()` to gather and display the information.

5.8.3 FUNCTION: (SKYMDS01P) INITIALISE

`initialise()` follows the usual pattern for skymds01 `initialise()` functions.

- catch termination signals
- read basic data from the environment
- initialise the curses system and display the window
- parse the single command line parameter – “insert/overwrite”
- get another couple of environment variables
- open all the necessary database tables

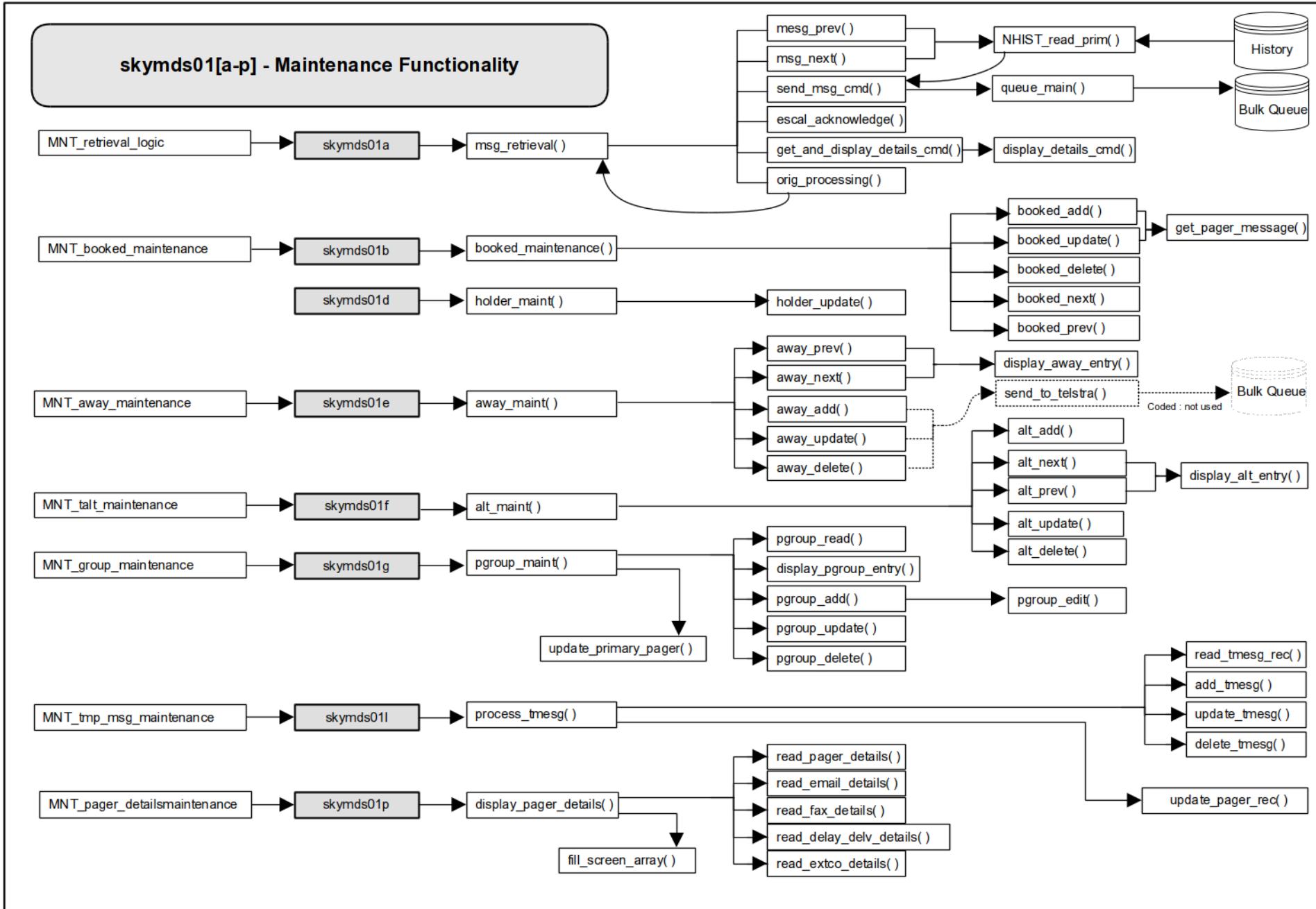


5.8.4 FUNCTION: (SKYMDS01P) DISPLAY_PAGER_DETAILS

`display_pager_details()` uses functions to collect all of the information it needs, **then** `fill_scrn_array()` writes this into the screen written by the `initialise()` function.

- `read_pager_details()` gets the necessary pager information
- `read_email_details()` gets the email address
- `read_fax_details()` gets the fax number
- `read_delay_delv_details()` to see when messages should be held up in the <System_Name> system
- `read_extco_details()` for the “Enhanced Service Details”
- finally `fill_screen_array()` writes this information to the screen

`display_pager_details()` now enters a holding loop, where the user can either press Shift-F11 to print the screen, or ESC back to from whence they came.



ONLY for personal, private use. Do not retain after use; do NOT distribute.

ONLY for use in association with hiring Stephen McGregor as a technical writer.

6 skymds08

6.1 SKYMD08 OVERVIEW

skymds08 is an interface to XACOM pbx/indial processor, used for processing touch-tone phone automatic pager alpha-numeric input. If, when a customer calls a <System_Name> centre, a computerised voice answers and the customer enters a series of digits to be sent to the pager, then skymds08 is the program managing these interactions.

6.2 RUNNING SKYMD08

The user replies ‘Y’ when prompted by the start-up script (/opt/mds/bin/start.mds) and skymds08 is run. Depending on demand a different number of copies of skymds08 are in the different cities, e.g. 5 in Sydney and 2 in Melbourne. skymds08 runs constantly as a daemon, not being spawned in response to an incoming call.

6.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-w	“What” mode: whether to send “Are you Alive” queries to the indial master
-T	Tandem mode
-D	Debug mode: listing detailed run-time information.
-c card_types	card_types is a list of card types (P or I) in use (“PPIIP...”)
-n num_cards	Number of active cards, or how many xp.rqc_reply.board_types[i] == 'I' fail before log it as an error
-t comms_tty	Port to talk to the Indial Box on.

Table 50: skymds08: Command-line Parameters

6.2.2 EXAMPLE COMMAND-LINE

A command-line used to run skymds08 in Sydney is:

```
$MDS_BIN_DIR/skymds08 -c PNNNNNNN -T -n 7 -t /dev/syb23 2>$LOG_SCR
```

The “2>\$LOG_SCR” appended redirects standard error (file descriptor 2) to what ever has been defined in the environment (or start-up script) the \$LOG_SCR - presumably a Log Screen.

6.3 PROGRAM DESCRIPTION

As opposed to running a “Process Message” function, skymds08 uses its main function to manage the most important attributes of its messages processing ; getting the messages, deciding what to do with them, and then doing it.

After calling its initialise() function, which, amongst other things puts the process into the background, skymds08 calls get_valid_message() to read (validated) message from the IVR (Interactive Voice Response) server into a global PABX comms record (as described in mdsxprec.h). Then it uses the value returned by the aforementioned get_valid_message() to decide what type of message this actually is, and then handles it appropriately. The choices it makes in handling the message are one of:

- either validate the indial number, and send a message to the bulk queue, (INDIAL_VALIDATION)
- or validate the pager number entered, and send a message to the bulk queue, (PAGER_VALIDATION)
- or validate the message, and send a message to the bulk queue, (PAGER_MESSAGE)
- or test the cards - are enough of them OK?? (WHAT_CARDS)
- or re-route the pager
- or substitute the pager for another one

This get message – handle it process runs in an infinite loop. The last action of each loop is a test to break out of the loop. As a debugging aid, the option (depending on the command line parameters) of polling the PABX cards in order to check that they are OK is the very last action of each loop.

6.4 PROGRAM STRUCTURE

6.4.1 SOURCES FILES

Source File	Description
skymds08.c	The main source file.
alarmrtn.c	Generic alarm routine.
gensubs.c	Contains various general purpose routines that are separately compiled.
isstart.c	makes an index in the C-ISAM database
mdscdr.c	Contains various general purpose routines that are separately compiled.
mdscomio.c	Various serial IO routines
mdsemqmain.c	Generic message email queuing routine.
mdsginit.c	A number of functions which perform commonly used initialisation code segments.
mdskbld.c	Contains functions to build and unbuild pager keys, and setup a dummy pager.
mdslog.c	Generic error logging.
mdsread.c	Line file read functions during Austel Number Change.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurtn.c	Generic message queuing routine.
mdsseck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssetup.c	This module contains routines for a program to obtain setup definitions from a text setup file.
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions which perform operations to do with time.
security.c	Various Security Routines.

Table 51: skymds08: Source Files

6.4.2 HEADER FILES

The following is a list of all the *.h header files used in skymds08

Header File	Description
gentypes.h	user defined types and 'booleans': user defined types
gettime.h	c time field definitions.
macros.h	user defined macros.: definitions & macros for common fns
mdsaprec.h	A-Party master file: Email A party file
mdscdrec.h	CDR data structure, just to keep the: CDR data structure.
mdsflsz.h	standard <System_Name> includes.: field len & definitions used by mds
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsparam.h	parameters used by pms system: parameter values used by pms.
mdsparec.h	Pager file – keep mdskbld.c happy: (PAG) Pager Master File
mdsporec.h	only for security levels: Ports security master file: Ports Record
mdspsrc.h	Password master file: fabian – Password File for security offsets -??
mdsquarec.h	Contacts Message queue file: Destination Message queue file
mdsserec.h	Security Group master file: Security Groups master file
mdsxprec.h	PABX comms layouts

Table 52: skymds08: Header Files

6.4.3 DATABASE TABLES

The following is a list of each database table used by skymds08

Database Table	Description	Header File
mdsline	Line	mdslirec.h
mdspager	Pager master	mdsparec.h
mdsports	Ports security	mdsporec.h
mdspassw	Password username	mdspsrc.h
mdssecur	Security codes	mdsserec.h
mdsaparty	Mdsaparty	mdsaprec.h
mdsacrec	Alt call details master	mdscdrec.h
mdscdrec	Call details master	mdscdrec.h
Queues	Description	Header File
mdsqxx.txt	Destination queues	mdsquarec.h

Table 53: skymds08: Database tables used

6.4.4 ENVIRONMENT VARIABLES REFERENCED

Environment Variable	Description
MDS_DATA_DIR	The directory in which <System_Name> stores its database tables and queues
MDS_BIN_DIR	Where the <System_Name> executables are
MDS_MACHINE_ID	What machine are we on, e.g. S for Sydney, P for Perth ...

Table 54: skymds08: Environment Variables Referenced

6.5 SKYMD08: MAJOR FUNCTIONS

6.5.1 FUNCTION: MAIN PART 1: INITIALISE AND GET MESSAGES

The first half of skymds08's main function does, as the title above suggests, two tasks. It calls initialise() (q.v.). Next, after starting an infinite loop, it calls get_valid_message() q.v. that reads a (PET format) message from the PABX into a global PABX Data record, tests the return value – a multi-valued flag – in a switch statement and then uses the appropriate case statement code to properly handle the message. See “main () part 2: Process the Received Message” below.

6.5.2 FUNCTION: INITIALISE

- Read the environment variables:
 - ~ MDS_DATA_DIR
 - ~ MDS_BIN_DIR
 - ~ MDS_MACHINE_ID
 - ~ Get the prefixes for each dial card by reading the environment variables INDIAL_CARD_PREFIX_??. However this environment variable doesn't seem to appear on any <System_Name> server.
- Parse the command line.
- Put the process into the background. Combined with the main function running in a infinite loop, this makes the program run as a user-level daemon.
- Handle or block a range of signals (Ctrl-C, alarms, debug-mode signal, kill).
- Open and set-up the communication ports
- A few final bits:
 - ~ ascertain the semaphore to be used as a shut-down message by this process
 - ~ access or set-up the shared memory to be used to store the how-long-since-last-activity timestamp
 - ~ run any city-specific code

6.5.3 FUNCTION: GET VALID MESSAGE(MSG LENGTH)

`get_valid_message()` sits in an infinite loop, reading from the serial port until something is received. Once a message does arrive, and after testing its attributes (too long, wrong format, reading error), `get_valid_message()` copies part of the message into a global PABX communications data record before doing some pre-processing of the record:

Command (message type)	pre-processing
2	A message to process. Nothing is actually done as the message is already in the global PABX communications record.
J	Set the command to PAGER_VALIDATION.
1	Set the command to INDIAL_VALIDATION
3	The command is not changed.
4	Pager re-routing
5	pager substitution
7	request card details

Table 55: skymds08: get valid message() pre-processing

Each of the command options extracts the checksum from the PABX communications record.. The command is passed back to the calling main function as the ‘message type’ (mess_type).

6.5.4 FUNCTION: MAIN PART 2: PROCESS THE RECEIVED MESSAGE

After a message has been gathered and pre-processed by `get_valid_message()`, the skymds08 main function continues with the handling of the message. There are 6 alternative handling procedures, one of which is selected based on the message type.

As listed above, the options are:

- either validate the indial number, and send a message to the bulk queue, (INDIAL_VALIDATION)
- or validate the pager number entered, and send a message to the bulk queue, (PAGER_VALIDATION)
- or validate the message, and send a message to the bulk queue, (PAGER_MESSAGE)
- or test the cards - are enough of them OK?? (WHAT_CARDS)
- or re-route the pager
- or substitute the pager for another one

- India Validation

First `validate_inidial()` is called, which extracts the inidial from the PABX communication record and performs a few tests: account expired, wrong type of account – before setting up the reply and a couple of technical adjustments. Finally a record is written to the Bulk Queue.

- Pager Validation

Similar to Indial Validation, except that when `validate_pager()` is called it tests the pager number to which the message is going to be sent instead. First the pager number prefix is processed, before the type of destination pager is queried and the pager message doctored appropriately. Upon return, the `skymds08` main function writes to the Bulk Queue.

- Pager Message

For an actual pager message, `validate_message()` is called, before a record is written to the bulk queue. `validate_message()` merely checks that all the characters of the message are valid, replacing '*' digits with '-'.

- Test the PABX cards

If a message of type “WHAT_CARDS” is received, skymds08 checks to see that enough cards are operational. The code is written such that there must be failings on two consecutive WHAT_CARDS tests before an alarm is raised: an alarm message is written to the Bulk Queue.

- Re-route the pager

Nothing is done in this program (skymds08) in response to a PAGER_RE_ROUTING message: the hander-function is empty

- Pager Substitution

As with pager re-routing, nothing is done in response to a PAGER_SUBSTITUTION message.

6.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

The log messages produced by mdsem01 are listed following:

6.6.1 MDS_LOG MESSAGES

the message, returned by get_valid_message(), is of an unrecognized type

invalid mess type=<number>

the pager number “MDSALARM” could not be located in the database

board_alarm(): MDSALARM not on pager file

when reading the serial port (for messages) an unrecognized (< 0) status was returned

invalid read status =<number>

A port is locked

**Open of < port > failed.
This program may already be running**

Couldn’t lock the port

**Lock of xacom comms port failed
This program may already be running**

couldn’t write in send_dummy_reply()

write() failed, wrote < number > of <number> bytes

standard log in send_dummy_reply()

INDIAL MASTER RESET received dummy reply sent

in validate_user()

< current_operator > is not authorised for paging

The usage instructions are also printed to mds_log (as well as stderr).

6.6.2 LOG MESSAGES WRITTEN TO STDERR

The Usage function writes the parameters to stderr. The usage function is called whenever a ‘?’ is passed on the command line, an invalid baud rate is specified

6.6.3 DEBUG MODE ONLY: LOG MESSAGES WRITTEN TO STDERR

At the start of each ‘Get Message – Process it’ loop, prints out the type of the current message

```
"main: mess_type=< message code character >
```

For each INDIAL VALIDATION type of message, write the pager number

```
INDIAL_VALIDATION:  
pager_num=< pager number >, mess_sent
```

For each PAGER VALIDATION type of message, write the pager number

```
PAGER_VALIDATION:  
pager_num=< pager number >, mess_sent
```

For each PAGER MESSAGE type of message, write the pager number

```
PAGER_MESSAGE:  
pager_num=< pager number >, mess_sent= < message-text >
```

At the end of the program, once the infinite loop has been broken, an elapsed time message is printed.

```
main( )- elapsed_time=< length of time >  
-----
```

As soon as a message is read, print it out. All control characters (ASCII code < 40) are converted to their octal equivalents.

```
get_valid_message( ) < Date + time > nbr_read= <length >  
< message >
```

If a message arrives that’s purpose is to test the PABX cards, the following is printed

```
Card_Info-command=< char code >, curr_card= <number > Time= < date time >, li= <Line number >,  
card_type_arr[ < number > ]= < char code >
```

If TEST_MODE has been defined, then fill an empty structure and print it out – for some reason.

```
test_struct[ < number > ].command=' ', line=' ',  
Initialise of test_struct is complete
```

The message is printed out if a dummy reply is sent. All control characters (ASCII code < 40) are converted to their octal equivalents.

```
< message >
```

If a negative acknowledge message is sent:

```
<NAK><CR>
```

If an “Are you alive?” message is sent to the PASX

```
are_you_alive? sent.
```

Every time a message is validated, it is printed out afterwards. As before, all control characters (ASCII code < 40) are converted to their octal equivalents – though there should be none as validate_message() should have removed them.

```
< message >
```

Every time an indial is validated,

the PABX line is written:

```
Li = < PABX line number >
```

and the message is dumped

```
"pa=< message structure >, status= < character code >, subscriber= <subscriber type >
```

and the reply text is also dumped, illegal characters converted to octal

```
< message >
```

Every time a pager is validated

the pager number is listed

```
xp.rqd.pager_num=< pager number >
```

and the message is dumped, illegal characters converted to octal

```
< message >
```

Every time a user is validated, pager details are printed
pager_prefix_len=%d, pager_prefix

If we need to poll the PABX, a message is written
"poll sent

7 mdsemgw

7.1 MDSEMGW OVERVIEW

mdsemgw is the <System_Name> email gateway daemon. Its task is to collect, manage and supply emails for mdsem01, mdsem03 and mdsem03, the A-Party, B-Party, and Bulk SMS email processing daemons. mdsemgw does this by writing messages to the mdsemqu queue, each message containing, among other things, the file path and name of a text-file into which mdsemgw has written the email. Once an email record has been written to the email queue, the processing daemons, mdsem01, mdsem02, and mdsem03 can process the email at their leisure.

mdsemgw is executed by sendmail, collecting sendmail's email messages on its stdin, checking the messages, before writing a record to mdsemqu.

7.2 RUNNING MDSEMGW

mdsemgw is spawned by sendmail each time it receives a pager, SMS, or Bulk SMS message. Establishing this relationship is a two-step process.

First, in the /etc/mail/mailertable file, lines such as

```
pagertest.<Company_Name>.com.au    pager:lnko.<Company_Name>.com.au
page.<Company_Name>.com.au        pager:lnko.<Company_Name>.com.au
sms.<Company_Name>.com.au        sms:lnko.<Company_Name>.com.au
bulksms.<Company_Name>.com.au     bulksms:lnko.<Company_Name>.com.au
```

<Company_Name> any emails addressed to the domains in the left column to the “pager”, “sms” or “bulksms” tag in the right-hand column. The domain address in the right hand column is looked up in the /etc/hosts/ file for resolution to a machine upon which sendmail is run. In this case it is the local (lnko) machine.

Next, in /etc/mail/sendmail.cf, under “Local and Program Mailer specification”, the ‘pager’ tag is associated with instructions as to what sendmail should do. The three relevant examples are:

```
Mpager, P=/opt/mds/bin/mdsemgw,
D=/opt/mds/bin F=DEMSU1ns7, R=40, S=10, U=mgr:<System_Name>,
M=2048000, A=mdsemgw -r $u -s $f -e P -n 100 -t 120 -D -q /var/mds/data/logs/emgw_qu.log

Msms,   P=/opt/mds/bin/mdsemgw,
D=/opt/mds/bin F=DEMSU1ns7, R=40, S=10, U=mgr:<System_Name>,
M=2048000, A=mdsemgw -r $u -s $f -e A -n 100 -t 120 -D -q /var/mds/data/logs/emgw_qu.log

Mbulksms, P=/opt/mds/bin/mdsemgw,
D=/opt/mds/bin F=DEMSU1ns7, R=40, S=10, U=mgr:<System_Name>,
M=2048000, A=mdsemgw -r $u -s $f -e B -n 1 -t 300 -D -q /var/mds/data/logs/emgw_qu.log
```

where the ‘pager’ tag is associated with the first block of instructions, the ‘sms’ tag with the second, and ‘bulksms’ with the third (the ‘M’s have special meaning to sendmail). The command line follows the ‘A=’

7.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-p <comms_port>	An optional port to associate with the program. If not used, the terminal from which mdsemgw is being run from is used, /dev/null if not attached to a terminal
-D	debug mode: listing detailed run-time information.
-r <recipient name>	The email address to which the email was sent
-s <sender address>	The sender's email address
-e <email type>	Where P=pager, A=sms, and B=Bulksms
-t <number of seconds>	The time within which frequency of emails is checked
-n <number of emails>	The number of emails that may be sent every -t <seconds>
-b <number of seconds>	How long to block an over-active sender for.
-q <log-file>	Write log messages to a file

Table 56: mdsemgw: Command-line Parameters

7.2.2 EXAMPLE COMMAND-LINE

From above, an example command-line used to execute mdsemgw (for SMS messaging):

```
mdsemgw -r $u -s $f -e A -n 100 -t 120 -D -q /var/mds/data/logs/emgw_qu.log
```

indicating that mdsemgw is to run:

- taking the sendmail destination address, held as the sendmail macro \$u, as the destination address
- taking the sendmail senders address, held as the sendmail macro \$f, as the senders address
- processing SMS messages. ('-e A')
- 100 emails can be received every 120 seconds from one originating email address ('-n 100 -t 120')
- in debug mode
- writing log messages to the file /var/mds/data/logs/emgw_qu.log

7.3 PROGRAM DESCRIPTION

mdsemgw is effectively a customised email processing utility for sendmail, being run with the appropriate parameters by sendmail every time an email is received. As shown above, there are three configurations: pager messages, SMS, and bulk SMS.

The main logic of the program is contained in its `main()` function (as opposed to control passing straight to another routine). After initialisation (`initialise()`), the `check_block_email_table()` function ascertains that the email sender is, in fact, allowed to use the <System_Name> system. Regardless of the return value of `check_block_email_table()`, mdsemgw connects the global shared memory via pointers to a shared memory block and calls `check_shm_queue()` to add the current sender to a list of ‘recent email senders’. `check_shm_queue()` then tests whether this sender has been sending too many emails recently – also blocking them if they, indeed, have. Finally the email is either blocked; a log messages and, maybe an email message issued to record the fact, or, `get_email_message()` writes the email out to a uniquely named¹⁴ text file for later processing, adding this filename plus some other relevant information to a new record in the email queue C-ISAM database table. mdsem01, mdsem02, or mdsms03 use these records to read the email from the appropriate text files.

¹⁴

The position – a number – of the email message in the email queue is embedded into filename of the created test file.

7.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into mdsemgw.

7.4.1 SOURCES FILES

Description	
gensubs.c	Contains various general purpose routines that are separately compiled.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
mdsemlog.c	Email gateway log functions
mdsginit.c	A number of functions which perform commonly used initialisation code segments.
mdslog.c	Generic error logging.
mdsqmmt.c	Queues file updates for transmission to other machines.
mdsshsm.c	General functions for Shared Memory
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions which perform operations to do with time.

Table 57: mdsemgw: Source Files

7.4.2 HEADER FILES

Header File	Description
gentypes.h	User defined types and 'booleans'.
gettime.h	C time field definitions.
macros.h	User defined macros. definitions & macros for common functions
mdscdr.h	CDR definitions file
mdsctype.h	Used for validating message chars.
mdsemrtn.h	Email routine.
mdsfldsz.h	Standard <System_Name> includes.: field length & definitions used by MDS
mdsparam.h	parameters used by PMS system

Table 58: mdsemgw: Header Files

7.4.3 DATABASE TABLES

The first tow of these contain a C-struct that matched the structure of the listed database table used. mdsemqu.h and mdsemqw.h are included because queues are kind of like database tables if you don't look too closely.

Database Table	Header File	Description
mdsparec.h	mdspager	Pager master file
mdpsrec.h	mdpassw	Password master file
mdsemgw.h	-	Email queue structures for mdsemgw
mdsemqu.h	mdsemqu	Email Queue file
mdsemblk.h	mdsemblk	Email Block file

Table 59: mdsemgw: Environment Variables

7.4.4 REFERENCED ENVIRONMENT VARIABLES

While mdsemgw is spawned by sendmail, and not run from start.mds, it adds the environment variables in the start.env file to its runtime environment. The environment variables used directly by mdsemgw are in the following table.

Environment Variable	Description
MDS_SHMIDS_DIR	The shared memory directory
MDS_DATA_DIR	Where the database tables and queues.
MDS_EMAIL_DIR	Emails logs and temporary files
MDS_LOG_DIR	Where log files are to be written to.
MDS_MACHINE_ID	The machine ID letter (e.g. 'S', 'X', 'M' ...)

Table 60: mdsemgw: Environment Variables

7.5 MDSEMGW: MAJOR FUNCTIONS

7.5.1 FUNCTION: MAIN

- Call `initialise()` to set-up the program's environment.
- Call `check_block_email_table()` to see if this sender is blocked
- Connect to a shared memory block listing the recent history of sent emails
- Call `check_shm_queue()`, which
 - ~ adds this email to the recent email list,
 - ~ checks whether too many emails have been sent from this address recently
- Either:
 - ~ Block the email, or
 - ~ call `get_email_message()` to write the email out to a file for later processing

7.5.2 FUNCTION: INITIALISE

The `initialise()` function sets up mdsemgw's run-time environment, via the following steps:

- open the system mail log file for sendmail
- Catch a whole range of signals
- Parse the command-line

See section 5.2.1 for a list of the command-line parameters and their meanings

- call `run_env_vars()` to add the environment variables in start.env to this program's environment
- Assign the used environment variables to global program variables. See section 5.5.4
- Open queues, log files and database tables:
 - email log
 - maintenance queue, for inter machine synchronisation
 - Pager Master C-ISAM database table
 - Username / password C-ISAM database table
 - email block C-ISAM database table
 - email queue C-ISAM database table.

Note that the email queue C-ISAM database table records contain the path/filenames of text files containing the emails, but not the emails themselves.

7.5.3 FUNCTION: CHECK_BLOCK_EMAIL_TABLE

First the domain portion of the sender's email address is looked for in the blocking table.

- If the domain isn't found then there can't be any block, so exit the function. Everything is OK.
- If the domain is there and no sender name, then one should block all emails from this domain, so block this email
- see if the block on this domain was added automatically by <System_Name> due to earlier emails from this domain arriving at a greater-than-allowed frequency. If so, see whether the mandatory blocking time has expired before blocking this email.

If both the domain *and* the sender name are present in the blocking file,

- if this address was added automatically by <System_Name> due to earlier emails arriving at a greater-than-allowed frequency, see if the mandatory blocking time has expired before blocking this email.
- Block the email.

7.5.4 CHECK_SHM_QUEUE

`check_shm_queue()` places the email's details in a queue of recently sent emails so that tracking of recently sent emails can be done (see 5.5.3 above), and flag whether this email's sender has sent to much email recently.

7.5.5 FUNCTION: GET_EMAIL_MESSAGE

This function writes the email to a text file, and writes a record containing the text file name to mdsemqu, the email queue.

First **get_email_message()** generates a file name from the time and the position in the email queue that the record containing this data will take – ensuring both a unique and relevant filename – following which the email(s) is/are written to a text file. Finally the record is actually written to the email queue database table (mdsemqu). This record contains, along with a couple of other things, the destination email address and the filename of the textfile containing the email text.

email addresses are in the form:

mobile_phone_number-PIN#@< destination domain >
e.g. 0421555111-9898@sms.<Company_Name>.com.au

where:

“mobile_phone_number” = the mobile phone number for the email to be sent to
“PIN#” = a four digit pin number.

destination domain = one of the four domains listed in the left hand column of the /etc/mail/mailertable file sample at the start of this section

filenames are in the form:

CCYYMMDDDHSS.aaaa
e.g. 200211030915.0042

where:

“CCYYMMDDDHSS” = represent the century (e.g. ‘20’ for 2002), year, month, day, hour, minute,
“aaaa” = the position in a (theoretically) 10,000 position email record queue.

7.5.6 LOGGING FUNCTIONS

mdsemgw uses the **log_msg()** function (coded in mdsemlog.c) to do its logging. **log_msg()** passes straight through to a **emlog_write()** function, which starts by checking that the email log file is current, and, if it isn’t opens a new one for today. After that it basically just writes out a time stamped log message. The log messages is written to a file named (something like)

MDS_EMAIL_DIR/logs/emgw020816.log,
(i.e. **/var/mds/data/email/logs/emgw020816.log**).

log_page() passes message straight through to **log_msg()**, while **log_if_debug** passes them through if the program is in debug mode.

The other logging method used is the macro MDS_LOG_DEBUG, defined in macros.h. This simple macro appends the file name and line number to the err_line before calling **mds_log()**, which writes this enhanced err_line to the file **MDS_DATA_DIR/mdslog.log**.

7.6 LOG MESSAGES PRODUCED

There are various ways to write a log message in a <System_Name> program. The simplest is send a message to stderr, which may then be duplicated to a file.

mdsemgw, and the other email processing programs – mdsem01, mdsem02, mdsem03, and mdssms01 – use their own logging routines contained in the file mdsemlog.c. The function called by mdsemgw is **log_msg()**.

7.6.1 WHILE IN DEBUG MODE ONLY

An email is blocked by the email block file

Block Email record matched

deleting the blocking record for this sender

Delete Block record [< sender >]

No environment variable specifying the shared memory directory

ERROR: get_env_variable() MDS_SHMIDS_DIR failed.

Shared Memory access failed

get_shm_id_key() failed

isrecurr() call failed in the C-ISAM database

isrecurr() < MDSEMBLK > failed

iswrcurr() call failed in the C-ISAM database

iswrcurr() < MDSEMblk > failed

iswrcurr() call failed in the C-ISAM database

iswrcurr() < MDSEMQU > failed

Log each email that is received

Read: < ropt_addr >, From: < sender_addr >

Telling a sender his/her email was blocked

Send Blocked Email back to sender

Noting the sender – domain split of an address

Sender = < sender > , Domain = < domain >

Shared Memory access succeeded

SHMKEY = < number >

7.6.2 ALWAYS DISPLAYED

Writing email to temporary text file

Write to < filename >

Couldn't read the email blocking email database table mdsemblk

Block Email Table isread() error =<err number >

message blocked

Blocked - Email Block record found

Could get shared memory key, but couldn't attach to the shared memory itself

Can't create shared memory

Could not open the temporary file for writing the email message to

Could not open < filename >

Created an email blocking record

Created Email Block record on [< sender >]

Created shared memory

Created EMGW shared memory. Size=< number >

No environment variable specifying the shared memory directory

ERROR: get_env_variable() MDS_SHMIDS_DIR failed.

Could not open the maintenance queue for communication between processes

mnt_queue_open() call failed

unknown signal received by the signal handling function

mdsemgw: Recv unknown signum=< number >, signal=< string >

Couldn't open the maintenance queue.

mnt_queue_open() call failed

Could not reopen the stdin for as the child's stdin after forking

send_mail(): dup2() failed.

Couldn't actually run send_mail

send_mail(): exec() failed

Couldn't open one end of the pipe used to communicate between the forked parent and child

send_mail(): fdopen() failed, errno= < err number >

Could not fork the mdsemgw process (en route to running one of the children as send_mail, in order to send an email)

send_mail(): pipe() failed, errno= < err number >

7.7 STOPPING MDSEMGW

mdsemgw is spawned by sendmail in response to each individual email arriving. After running it dies. Most standard signals that would have any adverse on the running of the program are converted to text and dumped, and no checking of the terminate semaphore takes place.

Basically, mdsemgw will die of its own accord, in its own good time.

8 mdsem01

8.1 MDSEM01 OVERVIEW

mdsem01 extracts B-Party SMS and Pager messages from the email queue – an Informix database table – processes them and adds them to the standard <System_Name> Bulk Queue¹⁵ for transmission to their destination – a mobile phone or a pager. B-Party messages are free to send, the cost being covered by the receiver: by a set charge, or on a per-message or per-character basis. B-Party are the standard form of SMS / Pager messages.

8.2 RUNNING MDSEM01

mdsem01 is started from the <System_Name> startup script /opt/mds/bin/start.mds. A single copy is run on each server, though, should volumes demand it, additional copies may be added.

8.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-t <comms tty>	Port to run on
-D	Debug mode: listing detailed run-time information.
-n number	Pause briefly if the size of the bulk queue is greater than < number >
-R	Request “return receipts” – ‘I got through’ messages
-c	Don’t check SMS security
-?	print the usage instructions
s	Stop if the size of the Bulk Queue grow greater than “s”. defaults to 90,000,000 i.e. don’t halt ¹⁶

Table 61: mdsem01: Command-line Parameters

8.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute mdsem01 at one <System_Name> site is:

```
$MDS_BIN_DIR/mdsem01 -R -t /dev/dum73 2>$LOG_SCR
```

indicating that

- return receipts are requested,
- the program is being run “on” port /dev/dum73
- and that stderr (file descriptor 2) is being redirected to LOG_SCR, defined in start.env to be /dev/tty0p5

8.3 PROGRAM DESCRIPTION

mdsem01 is a (background) daemon process that, as noted above, extracts B-Party SMS and Pager messages from the Informix email queue “mdsemqu”, processes the email contents before adding them to the Bulk Queue for transmission to their destination – a mobile phone or a pager.

The main function of mdsem01 consists of two function calls, `initialise()` which prepares the program and its environment, and `process_message()`, an infinite loop that, as the name suggests, actually undertakes the processing of the SMS and pager messages. The email subject line, the email text body plus any attached files are all converted into a Bulk Queue message for delivery to a pager or an SMS cell-phone.

¹⁵ Not to the SMS Bulk Queue. This is for A-Party messages only.

¹⁶ As of mid 2002, all <Company_Name>Net sites combined process about 30,000,000 messages a year. Even a most conservative estimate would set 90,000,000 bulk queue messages as at least a year’s messages.

8.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into mdsem01.

8.4.1 SOURCES FILES

Source File	Description
mdsem01.c	processed the MIME file and extract data sending it to bulkq queue.
mdsemenc.c	Contains functions which perform encoding message of files (up to 10 files) into ANSII and use sendmail to send mail out in a MIME format.
mdsemqmain.c	Generic message email queuing routine.
mdsemrtn.c	Contains functions which perform MIME operations (Multipurpose Internet Mail Extension)
alarmrtn.c	Generic alarm routine.
gensubs.c	Contains various general purpose routines that are separately compiled.
mdscdr.c	Contains various general purpose routines that are separately compiled.
mdsdecode.c	Decode MIME parts.
mdsextco.c	Functions to allow External User Interface (PET/MERP) to initiate enhanced services
mdsginit.c	A number of functions which perform commonly used initialisation code segments.
mdskbld.c	Contains functions to build and unbuild pager keys, and setup a dummy pager.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurt.n.c	Generic message queuing routine.
mdsseck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssetup.c	This module contains routines for a program to obtain setup definitions from a text setup file.
mdssubem.c	Generic function used in mdsem01, mdsem02, mdsem03
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions which perform operations to do with time.
mdsunixos.c	Unix OS parts.
mdsuudecode.c	Uudecode parts.
security.c	Various Security Routines.

Table 62: mdsem01: Source Files

8.4.2 HEADER FILES (NON-DATABASE)

Header File	Description
gentypes.h	User defined types and 'booleans'.
gettime.h	C time field definitions.:
macros.h	User defined macros.: definitions & macros for common fns
mdsflsz.h	Standard <System_Name> includes.: field len & definitions used by MDS
mdsparam.h	Parameter values used by PMS.

Table 63: mdsem01: Header Files

8.4.3 DATABASE HEADER FILES

Each of these header files contain (only) a C-struct that matched the structure of a database table used by mdsem01.

Database Table	Header File	Description
mdspager	mdsparec.h	(PAG) Pager Master File
mdspassw	mdpsrec.h	Password master file: fabian - Password File for security offsets
mdsports	mdsporec.h	only for security levels: Ports security master file: Ports Record
mdssecur	mdsserec.h	Security Group master file
mdsline	mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsconta	mdscorec.h	(CLT/INV/GRP) Contacts Master File: (CLT/INV/GRP) Contacts File
mdsqxx.txt	mdsqurec.h	Contacts Message queue file: Destination Message queue file
mdsaparty	mdsaprec.h	A-Party master file

Table 64: mdsem01: Database Tables Used

8.4.4 NECESSARY (REFERENCED) ENVIRONMENT VARIABLES

Environment Variable	Description
MDS_BIN_DIR	All the <System_Name> executables
MDS_DATA_DIR	The root of the data directories, queues, database and history.
MDS_EMAIL_DIR	the email spool file location
MDS_LOG_DIR	the directory for logs to be written to
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide B Brisbane P Perth C Canberra

Table 65: mdsem01: Environment Variables Referenced

8.5 MDSEM01: MAJOR FUNCTIONS

8.5.1 FUNCTION: MAIN

As noted above, the main function of mdsem01 consists of two function calls, `initialise()` which prepares the program and its environment, and `process_message()`, an infinite loop that, as the name suggests, actually undertakes the processing of the SMS and pager messages. The email subject line, the email text body plus any attached files are all converted into a Bulk Queue message for delivery to a pager or an SMS cell-phone.

8.5.2 FUNCTION: INITIALISE

The initialise() function sets up mdsem01's run-time environment, via the following steps:

- Reading Environment Variables

The environment variables as listed above are read in from the environment.

- Parse the command-line

As per 1.2.1 above, each command-line parameter is interpreted and mdsem01 adjusted appropriately. If in debug mode (i.e. the 'D' parameter passed to the program) a short, descriptive list of the command line options chosen is printed to stderr.

- `background()` function is called to put mdsem01 into the background.
- Set Signal handlers

Handlers for a large number of signals are set. Timeouts, debug signals and termination signals are specifically handled, a long list of other signals are directed to the `sig_dump()` function, which merely writes an information line to the MDS_ERR log file.

- Shut-down Semaphore, machine and operator Ids are established
- Open database tables as listed above
- Check the Port

Check that there is a valid record in the line database table for the port that was supplied

- Attach to the Security Shared Memory segment
- Open the email log file and the Bulk Queue

8.5.3 FUNCTION: PROCESS_MESSAGE

`process_message`, running in an infinite loop, strips the text from the email it is passed, converts it into an internal <System_Name> message and passes the message to the bulk queue.

- Start (or exit) an infinite loop

After setting up an infinite loop, the first thing `mdsem01` does is test the “`shutdown_semid`” semaphore, which, if set to zero, instructs the infinite loop to exit and the program, after doing a bit of clean-up, finishes.

- Get an email to process

The email queue database table ‘`mdsemqu`’ is read for any pending email messages. Now, a large amount of the time this table will be empty: all the email messages having been already processed. In this case `mdsem01` sleeps – by calling a special semaphore testing function with built-in sleeping – before ‘continue’-ing to the next iteration of the loop.

- Do some pre-processing and checking
 1. Strip the trailing spaces.
 2. Extract the pager number from the message
 3. Tests the permissions, based on the sender, the receiver, and what SMS carrier they may be using. A function `check_pager_and_security()` is used for this
 4. If this message is involved in an “Escalation” – advanced handling, or a “Roster” – timed changes to delivery, then print an error and dump the message. Escalations and roster messages are not to be sent via B-Party email.

- Get the content of the email

Any attached files are discarded, while the body of the message is read into a string, cleaned-up a bit and validated (checked for illegal characters).

- Prepare to post to the Bulk Queue

The first thing is to check the status of the Bulk Queue. If it is an unsuitable state then `mdsem01` exits. The other check is to prevent sending consecutive messages to the same pager being sent within one second of each other. `mdsem01` sleeps for a second if two such consecutive messages are sent.

- Post the message to the Bulk Queue

The message is sent to the bulk queue. There are two versions of this, one for ordinary messages and one for roster and contact (advanced handling) messages. If valid addresses have been supplied then send an “I’ve been sent” return receipt message back to the original sender.

- Write a “success” log message

Use the function `log_to_email_log()` to write a “success” log message.

8.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

The log messages produced by mdsem01 are listed following:

8.6.1 MDS_LOG MESSAGES

Trying to handle an unknown signal:

```
"mdsem01: Recv unknown signum=< signum >, signal=< signal-string >"
```

Can't find the shutdown_semid semaphore

```
"semget failed, errno=< error number >"
```

The first shutdown_semid is ignored with this message

```
"1st semctl failed, errno=< error number >"
```

This class of A-Party customer cannot page external contacts

```
"APARTY: Security <SMSx> not allow to page external contacts"
```

A port is supplied that doesn't appear in the mdsline database table

```
"sprintf(err_line, "isread( ) < text-string > failed, iserrno=< number >"
```

or

```
"The po.general_line_num < text-string > does not exist in < database table name >"
```

Can't parse a MIME message

```
"Error: extract MIME file=< file_path >, < opager number>, < sender address >"
```

Blank MIME file attached to an email

```
"Error: Extract data from MIME message is BLANK  
"num_att_files=< number of attached files>, file=< file path>, < receipt address >, < sender address >"
```

An A-Party Contact indial does not match pager's contact indial

```
"APARTY Contact indial [ indial number ] not match pager's contact indial [ indial number ]"
```

The setup for this A-Party customer does not allow a particular "Contact" advanced message handling behaviour

```
"APARTY ERROR: < details >"
```

log_to_email_log() writes a "success" log message once a message is added to the bulk queue.

8.6.2 LOG MESSAGES WRITTEN TO STDERR

SIGURG or SIGPIPE signal caught

```
fmdebug - got SIG=< number >
```

The usage instructions: a list of parameters – is printed if either an incorrect port or a '?' is supplied on the command line

8.6.3 DUBUG MODE ONLY: LOG MESSAGES WRITTEN TO STDERR

- A listing of the port setup after initialisation
- A raw dump of each message processed
- A detailed, itemised dump of each message processed
- "Get content of email" indicating that a message was successfully processed
- A detailed list of the destination pager details
- A short message noting that the process is going to sleep for one second due to two consecutive messages being sent to the same pager
- That a "contact" (Advanced Handling) message has been processed
- A print out of each return receipt used – twice
- Various A-Party set-up messages:
`"APARTY invalid Contact_operator permission"`
`"APARTY: Contact Service Funct Setup Incomplete"`
`"APARTY: Max concurrent Contact Services reached"`
`"APARTY Contact indial [indial] not match pager's contact indial[indial]"`
(also written to the mds_log)
- A couple of messages indicating that the Bulk Queue is over-full, and that mdsem01 is going to sleep for 5 seconds

8.7 STOPPING MDSEM01

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)

9 mdsem02

9.1 MDSEM02 OVERVIEW

mdsem02, running as a background daemon, extracts A-Party (paid for by the sender) SMS messages from the email queue database table (mdsemqu), processes these messages, and inserts them in the bulk queue.

One should note that the message itself isn't extracted from the mdsemqu queue, rather the file-name of the text file containing the message is retrieved. Also mdsem02 only processes SMS messages, while mdsem01, the B-Party email processor, produces both pager messages and SMS messages – as pager messages sent to a mobile phone.

mdsem03, the Bulk-SMS daemon, is an extension of the mdsem02 program.

9.2 RUNNING MDSEM02

mdsem02 is executed from each site's start.mds <System_Name> startup script.

9.2.1 COMMAND LINE PARAMETERS

Parameter	Description
<u>Mandatory</u>	
-d <queue>	The Port associated with this instance of the mdsem02. Its value is not used by mdsem02.
-i < string >	the destination queue name – a three or four capital letter name, not the two, lower case letter name. international SMS destination, e.g.: kaja
<u>Optional</u>	
-D	Run in Debug mode, listing detailed run-time information.
-n < number >	Sleep for 5 seconds if greater than < number > messages already in the bulk queue. default = 50
-s < number >	Exit if more than < number > messages already in the bulk queue. default = 90,000,000
-R	Send return receipt flags.
-c	Don't use this flag – obsolescent
?	Print the usage instructions

Table 66: mdsem02: Command Line Parameters

9.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute mdsem02 at one <System_Name> site (Perth) is:

```
$MDS_BIN_DIR/mdsem02 -R -i kaja -d SAU -t /dev/dum74 2>$LOG_SCR
```

indicating that mdsem02 is to run:

- sending return receipts (-R)
- the country codes ka and ja must be set up for this daemon to run
- the destination queue for the messages is SAU, which is the <Company_Name> SMS queue
- the program should be tagged as running on port /dev/dum74
- stderr is redirected to what-ever has been defined as LOG_SCR. This is defined as /dev/tty1p0 in start.env

9.3 PROGRAM DESCRIPTION

As noted above, mdsem02 extracts A-Party SMS messages from the mdsemqu queue and passes them to the bulk queue for delivery. It is very similar to mdsem01, that handles B-Party messages in the same manner.

The **main()** function calls **initialise()** to establish mdsem02's environment and put mdsem02 into the background; and **process_message()**, an infinite loop that, as the name suggests, actually undertakes the processing of the SMS and pager messages. The email subject line, the email text body plus any attached files are extracted from the textfiles in which they are stored, converted into a Bulk Queue message and posted to the bulk queue for delivery to a pager or an SMS cell-phone. **process_message()** is more involved in mdsem02 than in mdsem01, as in mdsem02 we have to undertake a series of checks to on the email sender, all of which are absent in mdsem01. These checks are detailed in section 7.5.3 below.

9.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into mdsem02.

9.4.1 SOURCES FILES

Source File	Description
alarmrtn.c	Generic alarm routine.
gensubs.c	Contains various general purpose routines that are separately compiled.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
mdscdr.c	Contains various general purpose routines that are separately compiled.
mdsdecode.c	Decode MIME parts.
mdsem02.c	processed the MIME file and extract data sending to bulkq queue for A PARTY SMS.
mdsemenc.c	Perform encoding of messages or files (up to 10 files) into an ANSI file and use sendmail to send mail out in a MIME format. MIME (Multipurpose Internet Mail Extension)
mdsemlog.c	Email gateway log functions
mdsemqmain.c	Generic message email queuing routine.
mdsemrtn.c	Contains functions which perform operations to do with MIME messages
mdsextco.c	Allow External User Interface (PET/MERP) to initiate enhanced services belonging to indials
mdsginit.c	A number of functions which perform commonly used initialisation code segments.
mdskbld.c	Contains functions to build and un-build pager keys, and setup a dummy pager.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for xmission to other machines.
mdsqurtn.c	Generic message queuing routine.
mdssemcck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssetup.c	This module contains routines for a program to obtain setup definitions from a text setup file.
mdssubem.c	Generic function used in mdsem01, mdsem02, mdsem03
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions which perform operations to do with time.
mdsunixos.c	Unix OS parts.
mdsuudecode.c	Uudecode parts.
security.c	Various Security Routines.

Table 67: mdsem02: Source Files

9.4.2 NON-DATABASE HEADER FILES

Header File	Description
gentypes.h	User defined types and 'booleans': user defined types
gettime.h	C time field definitions
macros.h	User defined macros.: definitions & macros for common fns
mdsfldsz.h	Standard <System_Name> includes.: field len & definitions used by MDS
mdsparam.h	parameters used by PMS system:

Table 68: mdsem02: Header Files

9.4.3 DATABASE TABLE

Each of these header files contain (only) a C-struct that matched the structure of a database table used by mdsem02.

Database Table	Header File	Description
mdsaparty	mdsaprec.h	Mdsaparty
mdsconta	mdscorec.h	Contacts master
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdspassw	mdspsrc.h	Password username
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsqrec.h	Destination queues

Table 69: mdsem01: Database Table used

9.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description
MDS_BIN_DIR	All the <System_Name> executables
MDS_DATA_DIR	The root of the data directories, queues, database and history.
MDS_EMAIL_DIR	the email spool file location
MDS_LOG_DIR	the directory for logs to be written to
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide B Brisbane P Perth C Canberra

Table 70: mdsem01: Environment Variables

9.5 MDSEM02: MAJOR FUNCTIONS

9.5.1 FUNCTION: MAIN

As noted above, the `main()` function in mdsem02 calls only two functions:

- `initialise()`, which sets up the program that the program is running in
- `process_message()`, which, in a loop reads the names of email-message-containing files and, optionally, process these files (i.e. the email messages)

9.5.2 FUNCTION: INITIALISE

The `initialise()` function sets up mdsem02's run-time environment, via the following steps:

- Reading Environment Variables
- Parse the command-line. As per 7.2.1 above
- Put the process in the background

the `background()` function is called to put mdsem02 into the background. This short function, coded in mdsginit.c, simply forks the process and kills the parent, leaving the child running in the background.

- Set Signal handlers

Handlers for a large number of signals are set. Timeouts, debug signals and termination signals are specifically handled, a long list of other signals are directed to the `sig_dump()` function, which merely writes an information line to the MDS_ERR log file.

- Set the operator ID, the shut-down semaphore and the local Machine ID
- Open database tables

Opens: mdsaparty, mdsconta, mdsline, mdspager, mdspassw, and mdsparts – as described above

- Check that there is a valid record in the line database table for the port that was supplied
- Open the email log file and the Bulk Queue

9.5.3 FUNCTION: PROCESS_MESSAGE

`process_message()` is the main work function of mdsem02. Running in a loop, it tests the information contained in each email queue record and, if the tests are passed, reads the file (i.e. the email) from disk, does a couple of tests on the email message itself, before sending the email text as a message to the bulk queue. The destination email address and the name of the text file containing the email body are contained in the mdesmqu records.

As discussed in section 5.5.5 (mdsemgw):

email addresses are in the form: mobile_phone_number-PIN#@< destination domain >
e.g. 0421555111-9898@sms.<Company_Name>.com.au

filenames are in the form: CCYYMMDDHHSS.aaaa
e.g. 200211030915.0042

process_messages() passes through the following steps:

- start the infinite loop
- test the ‘shut down’ semaphore – Shall I shut down??
- Read and test and email record from the email queue,
- test that the email type recorded is appropriate

If the email type is not appropriate, go to the next database record. The appropriate email type for mdsem02 is ‘A’, which means “SMS Message”¹⁷.

- Next test the phone number (mobile_phone_number-PIN# - e.g. 0421555111-9898) embedded in the destination email address:
 1. extract the destination phone number from the email address
 2. if an international number, test that the international code is correct / possible
 3. check the destination number in an appropriate format
 4. check that the sender and destination address both satisfy A-Party security requirements
 5. optionally: if we need to check the email gateway, check it
 6. check the PIN number
- if all the checks above are passed, call **get_email_files()** to access the email inside the textfile.
- check the type of content of the email and its attachments.

MIME_check_msg_type(), (in mdsemrtn.c) ascertains the content type of the email. At the time of writing MIME types 1 (text, i.e. no attachment), 6 (multi-part), 7 (message-part) and 8 (uuencode¹⁸) are supported by mdsem02. If the content is not supported, or there is an attachment (MIME type = 6,7, or 8), but it is empty, then delete the file and database record, and go on to waiting for the next email.

- Read the email itself.

MIME_get_basicpart() (also in mdsemrtn.c) actually reads in both the basic-part (the body) of the email *and* all the attachments into a mime_message structure

- Process the message

Most of the checks have been done. Just validate the message, checking for illegal characters, and stripping any trailing blanks

- Post the message to the bulk queue

Check the bulk queue, and call **send_messages()** to send the email and each attachment separately as a message to the bulk queue.

- Send a Return Receipt if required.
- Clean up

Clean up this message, write a log message and return to waiting for the next A-Party SMS message.

¹⁷ The variable in the code is EMQU_TYPE_ASMS, which is set in mdsemqu.h, the header file for the email queue, to ‘A’. In mdsem01, the appropriate email type is ‘P’, which means both pager messages and SMS messages – the SMS messages actually being pager messages sent to your cellphone.

¹⁸ uuencode is a method of encoding binary data as text

9.5.4 OTHER FUNCTIONS

9.5.4.1 get_email_files

In mdssubem.c, `get_email_files()` fills the `MIME_Record` structure displayed in the appendix to this section. This structure contains all the information about the file(s) that contain(s) the email's contents.

9.5.4.2 MIME_get_basicpart

After being passed a structure generated by `get_email_files()` `MIME_get_basicpart()` reads the email and the attached files from the text-files into which they have been written by mdsemgw, and writes out their contents to one long string.

9.5.4.3 Send_Messages

In mdssubem.c, `send_messages` cycles through a string made by `MIME_get_basicpart()` and converts each of the email messages and former attachments embedded in the string into a message for insertion onto the bulk queue. If the `write_bulkq = 0`, the messages are posted to a special email queue, else, as here in mdsem02, they are written straight to the bulk queue.

9.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

The log messages produced by mdsem01 are listed following:

9.6.1 MDS_LOG LOG MESSAGES

`mds_log()`, defined in mdslog.c, writes error messages to both:stderr, and `MDS_DATA_DIR\mdslog.log`

The email gateway is not specified in the MIME (document) header

Error: gw_domain=< domain > - not find in MIME header

mdsem02 can only process text files, text files with attachments and uuencoded files. Other MIME type produce this error

Error: not support content type=< type number >, file=< file path >, <destination address>, < sender address >

Return email address in an invalid format

Error: ropt_addr=< Cell-phone number > - Invalid format ,ropt_addr

The PIN number supplied, embedded in the 'to:' email address, is invalid

Error: sender=< email address >, ropt=< > - Invalid PIN

One of the country codes (pairs of letters) does not appear in the database

Int. SMS destinations list do not exist in DB

On the command line a series of pairs of letters are passed to mdsem02 indicating which countries can be internationally SMS messaged. If the string of pairs is too long the this error is issued

The limit for the INTERNATION SMS range is < > digits,

The list of pairs of characters must have an even-numbered length

There must be an even number of characters entered for

9.6.2 MDS_LOG_DEBUG LOG MESSAGES

MDS_LOG_DEBUG() a macro wrapper for **mds_log()**, appends the Filename and Line number to the error message, before passing it to **mds_log()** as described above. Thus **MDS_LOG_DEBUG()** messages are also duplicated to both stderr and **MDS_DATA_DIR/mdslog.log**. **MDS_LOG_DEBUG()** is defined in macros.h

Setting the shutdown_semaphore failed

1st semctl failed, errno=< Error Number >,

The bulk queue has grown to greater than the limit set. The program is exiting

Bulkq filesize= < queue size > Filesize limit= < queue limit size >. Exiting

There are no attachments, when it is thought that there should be

Error: Extract data from MIME message is BLANK

An error occurred when trying to open the text-file containing the email

Error: extract MIME file=< file-path >, < cell-phone number >, < sender's email>

The line number in the port record isn't valid. This indicates that an incorrect port (/dev/abcxyz) has been passed on the command line

isread() < MDSLIN > failed, iserrno=< iserrno >

Unknown signal received in the signal handling function

mdsem01: Recv unknown signum=< number >, signal=< signal >,

Could not get the shutdown_semaphore during start-up

semget failed, errno=< errno >,

The line number in the port record isn't valid. This indicates that an incorrect port (/dev/abcxyz) has been passed on the command line.

The po_general_line_num < line > does not exist in <mDSLIN - database table>

9.6.3 STDERR – DEBUG MODE ONLY. LOG MESSAGES

Whether or not one is running mdsem02 in debug mode, the usage instructions are dumped if incorrect command-line parameters are passed, or a ‘?’ is passed on the command-line.

Caught a signal by a signal handler

firebug - got SIG < signal number >

a dump of command line parameters

Port :	< port string >
bulkq_fsize_limit:	< limit set on the bulk queue >
queue_threshold:	< sleep-for-5-secs queue size >
return receipt :	Whether to send return receipts

a dump of each email data record.

Record= < email data record >

a dump of each A-Party client

-----'A' party Details -----	
Domain Name :	< Domain Name part of senders address >
Sender Name :	< Sender's name part of sender's email address >
Gateway Domain:	< Gateway Domain: what <Company_Name> domain the email was sent to >
Aparty ID :	< ID of the A-Party client >
Client Number:	< Client Number >
PIN :	< PIN supplied >
Client Name :	< Name of A-Party client >
Priority Flag:	< Priority level >
Status :	< status (active, suspended ...) >

The email gateway is not specified in the MIME (document) header

Error: mdsemgw_domain=< domain > - not find in MIME header

A dump of details of each attached file

```
mime_file: < File containing basic email >
content_type: < basic email content type > (should be '1')
nbr_att_files (inc basicpart): < number of attached files >
=====
| Idx | Filesize | Type | Filename |
=====
[for each attached file ]
< filesize >, <content type code (e.g. 1,3, or 7) >, < filename >
< filesize >, <content type code (e.g. 1,3, or 7) >, < filename >
```

Getting the content of the email

Get content of email

What is about to be added to the bulk queue

*******QUEUE DETAILS START*******

Page

6

< Home city of client – inherited from paging >

Page FunctDig

1

Message

1

Message : < the message >

~~-----~~

*****QUEUE DETAILS END*****

Return receipt

cnt unsent bulk= < number >

Bulk queue is backing up - sleep
 \geq bulk threshold \leq min

Sleep for < number > seconds

sleep for < number > seconds

9.7 STOPPING MDSEM02

1. Set the shutdown_semid to non-zero.
 2. Find its PID and KILL -9 it (as root)
 3. If, when checking the Bulk Queue, the status returned is ‘EXIT’, then mdsem02 will terminate.

9.8 APPENDIX TO MDSEM02 – MIME RECORD STRUCTURE

The following is a representation of the structure used to reference a file or files containing an email or an email and its attachments in <System_Name>.

- Content Type (0 to 9)
- Number of Attached Files (0 to 10)
- Header:
 - from Address
 - to Address
 - Return Receipt Address
 - Subject line
- Array of MIME_MAX_FILES (= 10) of:

filename	1
filesize	
content type	
encoding type	
filename	2
filesize	
content type	
encoding type	
filename	3
filesize	
content type	
encoding type	
:	:
:	
:	
:	
filename	MIME_MAX_FILES (= 10)
filesize	
content type	
encoding type	

10 mdsem03

10.1 MDSEM03 OVERVIEW

mdsem03, the Bulk-SMS daemon, produces collections of Bulk-SMS messages, the details of which have been pre-established by (the attachments to) messages sent to the mdsemgw daemon. Running as a background daemon, mdsem03 extracts the file locations of A-Party¹⁹ Bulk-SMS emails from the email queue (database table mdsemqu), processes the *attachments* in these emails, and inserts the generated SMS messages them into one of the four ‘priority’ SMS queues. Another program, mdsem01, extracts the messages from these priority queues and posts them to the Bulk SMS Bulk Queue for eventual delivery. Bulk-SMS messages are SMS messages sent to a number²⁰ of cellphones at once.

Please note that:

- it is the file-name of the text file containing the email message – not the message itself – held in the mdsemqu queue record. See Section 7 (mdsem02), Appendix 1 for a representation of the structures retrieved from the email queue.
- the email messages must contain attached files. It is these attached files that contain the instruction for the generation of the Bulk-SMS messages.

mdsem03 is a bulk SMS extension of mdsem02, the email-to-SMS daemon. mdsem02 is described in section 7.

10.2 RUNNING MDSEM03

mdsem03 is executed from each site’s start.mds <System_Name> startup script.

10.2.1 COMMAND LINE PARAMETERS

Parameter	Description
Mandatory	
-d <queue>	The Port associated with this instance of the mdsem03. Its value is not used by mdsem03.
-i <string>	the destination queue name – a three or four capital letter name, not the two, lower case letter name. international SMS destination, e.g.: bsja
Optional	
-D	Run in Debug mode, listing detailed run-time information.
-m	The maximum number of messages that can be generated BulkSMS file. default = 10,000
-n <number>	Sleep for 5 seconds if greater than <number> messages in the bulk queue. default = 50
-s <number>	Exit if more than <number> messages already in the bulk queue. default = 90,000,000
-R	Send return receipt flags.
-c	Don’t use this flag – obsolescent
?	Print the usage instructions

Table 71: mdsem03: Command-line Parameters

10.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute mdsem03 at one <System_Name> site (Adelaide) is:

```
$MDS_BIN_DIR/mdsem03 -R -i bsja -m 50000 -d BULKS -t /dev/dum75 2>$LOG_SCR
```

indicating that mdsem03, found in the MDS_BIN_DIR directory, is to run:

- sending return receipts (-R)
- the country codes bs and ja must be set up for this daemon to run
- up to 50,000 SMS messages can be generated from one SMS text file attached to an email
- the destination queue for the messages is BULKS, which is the Bulk-SMS queue
- the program should be tagged as running on port /dev/dum75
- stderr is redirected to what-ever has been defined as LOG_SCR. This is defined as /dev/tty1p0 in start.env
- \$MDS_BIN_DIR is generally defined (in start.env) to be ‘/opt/mds/bin’

¹⁹ paid for by the sender

²⁰ up to 50,000. Small numbers of SMS messages in a “Bulk” SMS send are possible, e.g. 5 SMS messages to one’s five associates.

10.3 PROGRAM DESCRIPTION

mdsem03 extracts the file locations of A-Party Bulk-SMS email messages from the email queue (database table mdsemqu), processes these messages, and inserts the generated SMS messages them into one of the four ‘priority’ SMS queues. Another program, mdsms01, then extracts the messages from the four priority queues for insertion into the Bulk Queue. mdsem03 is a Bulk-SMS extension of mdsem02.

mdsem03’s **main()** function makes only two calls:

- **initialise()** – to set up the program and its environment, and
- **process_message()** – which, in an infinite loop, converts each correctly formatted email attachment into a, possibly large, number of SMS messages.

Actually, there is a third call: **finalise()** is used to clean up when mdsem03 exits.

process_message() checks the details of each email (held as a text-file), the email itself, and finally uses **check_file_content()** on each attachment of the email. Upon passing each of these tests, a call is made to **process_bulksms_file()** to generate SMS messages from (each of) the Bulk-SMS instruction files appearing as the email’s attachment(s).

10.3.1 LOGIC OF A BULKSMS INSTRUCTION FILE

BulkSMS instruction files contain lines consisting of:

- An initial tag, or
- text, without an initial tag, or
- blank lines

The tags are: <#UNIQUEID#>, <#BULKMSG#>, <#PRIORITY#>, and <#BULKID#>, while text without tags is data relevant to the previous tag. Note that, as each destination mobile-phone number has to be included, this may be a large file.

10.3.1.1 A Bulk SMS instruction file example

```
<#UNIQUEID>
0411123456
BATCH NUMBER 445637 COMMENCING

<#BULKMSG#>
4 BEDROOM HOME JUST LISTED - PLS
CALL ABC REAL ESTATE

<#PRIORITY#> 2

<#BULKID#>
0411399483
0412334562
0455876223
0411410444
<#PRIORITY#>
1

<#BULKMSG#>
AUCTION THIS SATURDAY FOR 24 SMITH ST HAS BEEN CANCELED

<#BULKID#>
0477885377
0411334564
0422334093
<#UNIQUEID#>
0411123456
BATCH NUMBER 445637 COMPLETED
```

10.3.1.2 Bulk-SMS instruction file tags

- <#UNIQUEID#> The UNIQUEID indicates that the following text spells out the information required for an entire, single SMS message. In this example, the UNIQUEID tag is used to send messages back to the originator's phone, indicating that their BulkSMS messages have been sent. Any SMS message could be sent here, as the text after a UNIQUEID is processed in the same way as any other SMS message.
- <#BULKMSG#> The BULK-SMS message that will be sent to multiple SMS destinations.
- <#BULKID#> The destination phone numbers
- <#PRIORITY#> Each A-Party client has a default priority written in their mdsaparty (A-Party) database table. If a SMS-Instruction file uses this tag it overrides the default priority. 1 is the highest priority, 4 the lowest.

10.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into mdsem03.

10.4.1 SOURCES FILES

Source File	Description
mdsem03.c	processed the MIME file and extract data sending to priority queue for 'A' Party BulkSMS.
alarmrtn.c	Generic alarm routine.
gensubs.c	Contains various general purpose routines that are separately compiled.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
mdscdr.c	Contains various general purpose routines that are separately compiled.
mdsdecode.c	Decode MIME parts.
mdsemenc.c	perform encoding of messages or files into a ANSI file and to send mail out in a MIME format.
mdsemlog.c	Email gateway log functions
mdsemqmain.c	Generic message email queuing routine.
mdsemrtn.c	Contains functions which perform operations to do with MIME
mdsextrco.c	Functions to allow External User Interface (PET/MERP) to initiate enhanced services
mdsginit.c	A number of functions which perform commonly used initialisation code segments.
mdskbld.c	Contains functions to build and unbuild pager keys, and setup a dummy pager.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqrtn.c	Generic message queuing routine.
mdssemcck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssetup.c	This module contains routines for a program to obtain setup definitions from a text setup file.
mdssubem.c	Generic function used in mdsem01, mdsem02, mdsem03
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions which perform operations to do with time.
mdsunixos.c	Unix OS parts.
mdsuudecode.c	Uudecode parts.

Table 72: mdsem03: Source Files

10.4.2 NON-DATABASE HEADER FILES

Header File	Description
gentypes.h	User defined types and 'booleans': user defined types
gettime.h	C time field definitions.
macros.h	User defined macros.: definitions & macros for common functions
mdsaprec.h	Email A party file
mdscdr.h	CDR definition files
mdscdrec.h	CDR data structure
mdscorec.h	(CLT/INV/GRP) Contacts Master File
mdsemqu.h	Email Queue file
mdsemrtn.h	Email routine.
mdsextco.h	Header file used for the External Contacts
mdsflpsz.h	Standard <System_Name> includes.: field len & definitions used by MDS
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsparam.h	Parameter values used by PMS.
mdsparec.h	Pager Master File
mdsporec.h	only for security levels: Ports security master file: Ports Record
mdspsrc.h	Password master file
mdsqrec.h	Queue header.
security.h	security functions: Structure and defines for security groups

Table 73: mdsem03: Header Files

10.4.3 DATABASE TABLE

Each of these header files contain (only) a C-struct that matched the structure of a database table used by mdsem03.

Database Table	Header File	Description
mdsaparty	mdsaprec.h	Mdsaparty
mdsacrec	mdscdrec.h	Alt call details master
mdscdrec	mdscdrec.h	Call details master
mdsconta	mdscorec.h	Contacts master
mdsdestm	mdsderec.h	Destinations – countries, cities and networks
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdspports	mdsporec.h	Ports security
mdspassw	mdspsrc.h	Password username
mdsqxx.txt	mdsqrec.h	Destination queues – Standard queue header.

Table 74: mdsem03: Database Table used

10.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description
MDS_BIN_DIR	All the <System_Name> executables
MDS_DATA_DIR	The root of the data directories, queues, database and history.
MDS_EMAIL_DIR	the email spool file location
MDS_LOG_DIR	the directory for logs to be written to
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide B Brisbane P Perth C Canberra

Table 75: mdsem01: Environment Variables

10.5 MDSEM03: MAJOR FUNCTIONS

10.5.1 FUNCTION: MAIN

As noted above, the **main()** function in mdsem03 calls only two (well, three) functions:

- **initialise()**, which sets up mdsem03.
- **process_message()**, which, in a loop, reads the names of email-message-containing files and, optionally, process these files to produce (Bulk) SMS messages.
- also **finalise()**, which closes all open files and database tables.

10.5.2 FUNCTION: INITIALISE

The **initialise()** function sets up mdsem03's run-time environment, via the following steps:

- Reads and processes Environment Variables, copying them into program variables. See 8.4.4.
- Parse the command-line. As per 8.2.1 above
- Put the process in the background

the **background()** function is called to put mdsem03 into the background. This short function, coded in mdsginit.c, simply forks the process and kills the parent, leaving the child running in the background.

- Set Signal handlers

Handlers for a large number of signals are set. Timeouts, debug signals and termination signals are specifically handled, a long list of other signals are directed to the **sig_dump()** function, which merely writes an information line to the MDS_ERR log file.

- Set the operator ID, the shut-down semaphore and the local Machine ID
- Open database tables

Opens: mdsaparty, mdsconta, mdsline, mdspager, mdspassw, and mdsports – as described above

- Check that there is a valid record in the line database table for the port that was supplied
- Open the email log file and the all the email queues.

10.5.3 FUNCTION: PROCESS_MESSAGE

process_message() is the main work function of mdsem03. Running in a loop, **process_message()** checks the details of each email (held as a text-file), the email itself, and finally uses **check_file_content()** on each attachment of the email. Upon passing each of these tests, a call is made to **process_bulksms_file()** to generate SMS messages from (each of) the email's attachment(s). The destination email address and the name of the text file containing the email body are contained in the mdesmqu records.

10.5.3.1 Similar to mdsem02, the single-SMS daemon

mdsem03's **process_message** is a Bulk SMS extension to mdsem02's **process_message()**, sharing much of the same code. Also, as discussed in section 5.5.5 (mdsemgw):

email addresses are in the form: batch_Number-PIN#@< destination domain >
e.g. 112233-9898@bulksms.<Company_Name>.com.au

filenames are in the form: CCYYMMDDHHSS.aaaa
e.g. 200211030915.0042

`process_messages()` passes through the following steps:

- start the infinite loop
- test the ‘shut down’ semaphore – Shall I shut down??
- read an email record from the email queue. If there is nothing in the queue mdsem03 sleeps by calling `sem_check()`, which (via a third function) puts the process to sleep for `sleep_time(5)` seconds.
- test that the email type recorded is appropriate

If the email type is not appropriate, go to the next database record. The appropriate email type for mdsem03 is ‘EMQU_TYPE_BULKSMS’ which is #defined in mdsemqu.h to ‘B’, which, in turn, means “Bulk SMS Message”.

- Next test the Batch number an PIN number embedded in the ‘To’ email address of the email originally sent to the <System_Name> site. All emails, to any address @bulksms.<Company_Name>.com.au will get this far (unless you included a movie as an attachment or something silly like that). Here, only emails with a correct Batch number and a correct PIN number in the address are retained – all others are deleted.
 1. Check the Batch Number
 2. Check that the sender address satisfies the A-Party security requirements
 3. optionally: if need to check the email gateway, check it.
 4. check the PIN number
- if all the checks above are passed, call `get_email_files()` to fills the `MIME_Record` structure with information about the files that contain the email’s contents.
- check that there is an attachment.
- `MIME_check_msg_type()`, (in mdsemrtn.c) ascertains the content type of the email.

At the time of writing MIME types 1 (text, i.e. no attachment), 6 (multi-part), 7 (message-part) and 8 (uuencode²¹) are supported by mdsem03, but as all BulkSMS messages must have an attachment, only 6,7, and 8 are acceptable. If the content is not supported, or there is an attachment but it is empty, then delete the file and database record, and go on to waiting for the next email.

10.5.3.2 Specialised Bulk SMS processing

- check that the first attachment is a text-file, or the whole message is uuencoded, or the first attached file with a *.txt extension has a `MIME_file_type` of `MIME_CONTENT_TEXT` (i.e. 1).

Only one attached file will actually be processed by `process_message()`. This step locates that file from amongst the email’s attachments.

- `MIME_get_att_file()` actually retrieves the chosen file, that is, returns a FILE pointer to the open file on disk.
- `check_file_content()` ensures that the contents of the text file make sense as a BulkSMS instruction file.
- Next, check that the number of messages that are going to be produced is not greater than the maximum allowed.
- Finally `process_bulksms_file()` generates the messages and inserts them into the priority queues.
- Send a return-receipt message to the sender

These final two parts of `process_message()` are again the same as mdsem02.

- return to looping to wait for the next Bulk SMS message to arrive.

²¹ uuencode is a method of encoding binary data as text

10.5.4 FUNCTION: PROCESS_BULKSMS_FILE

[See the Bulk-SMS instruction file sample in section 8.3.1.1 above]

`process_bulksms_file()` reads each line of a BulkSMS instruction file, noting and acting on each tag and its accompanying text, generating SMS messages, and posting them to one of the four priority queues.

- Store position for later recovery if a problem happens part-way through processing a BulkSMS instruction file.
- Call `check_tag_state()` to read the tag (if any) from the current line. The return value is a program variable holding a representation of the tag, or UNKNOWN_STATE if there is no tag on the line.

10.5.4.1 Processing the <#BULKMSG#> tag: The SMS message Text

This tag flags the SMS message text, and is represented internally as ‘MESG_ID’.

1. if the previous tag was the UNIQUEID tag, then this indicates the end of the text associated with that tag. So call `check_unique_msg()` to convert the text collected into a <System_Name> format message before using `process_bulksms_msg()` to post it to a priority queue
2. Append the SMS message text to any existing text for the current message.

Continuation: Append the SMS message text to any existing text for the current message.

10.5.4.2 Processing the <#BULKID#> tag: the destination mobile-phone numbers

The BULKID tag is used to record the destination phone numbers for the SMS messages.

1. if the previous tag was the UNIQUEID tag, then this indicates the end of the text associated with that tag. So call `check_unique_msg()` to convert the text collected into a <System_Name> format message before using `process_bulksms_msg()` to post it to a priority queue.
2. Read the phone-number, test that it is only made of numbers, and re-format the cellphone number.
3. Post a SMS message, containing the text collected so far, to one of the four priority queues.

Continuation: Steps 2 and 3, process this phone number and send the SMS message text to this phone number

10.5.4.3 Processing the <#PRIORITY#> tag

The PRIORITY tag changes the priority of the message handling from the default priority set for the current A-Party customer (in the A-Party database table: mdsaparty), to the priority set by the single digit number (1,2,3, or 4) following the PRIORITY tag. The priority represents which of the four Bulk-SMS queues the message is delivered to. Queue 1 is the highest priority. 2,3, and 4 are of a lower – and in reality very similar – priority. All priority 1 messages are processed before a small number of messages from the other queues, in order (2: 3: 4) are delivered, before all the queue 1 messages are then processed again.

1. The first numeric digit following the PRIORITY tag is set as the priority – as long as it is between 1 and 4.
2. if the previous tag was the UNIQUEID tag, then this indicates the end of the text associated with that tag. So call `check_unique_msg()` to convert the text collected into a <System_Name> format message before using `process_bulksms_msg()` to post it to the appropriate priority queue.

Continuation: There should be no continuation of the text following this tag, so continuations are ignored.

10.5.4.4 Processing the <#UNIQUEID#> tag

The UNIQUEID tag contains the complete data required for a single SMS message. In the Bulk-SMS instruction file example above this tag is used to send messages back to the Bulk-SMS originator to inform him or her that the SMS messages have been delivered. However, there is no reason for this tag to be used for this purpose only.

1. if the previous tag was the UNIQUEID tag, then the appearance of another UNIQUEID tag indicates the end of the previous UNIQUEID tag. So call `check_unique_msg()` to convert the text collected from the previous UNIQUEID tag into a <System_Name> format message, before using `process_bulksms_msg()` to post that message to a priority queue.
2. copy the SMS message text following this tag into a buffer.

Continuation: Send the collected text to a priority queue as a SMS message. At the end of the text following the UNIQUEID tag – i.e. the start of the next tag – the collected UNIQUEID text is processed by the code for each of the tags detailed above (*q.v.*).

10.5.5 FUNCTION: CHECK_FILE_CONTENT

check_file_content(), as the name suggests, checks that the Bulk-SMS instruction file is in the correct format. While this is quite a long function, it is, basically, a practice run of **process_bulksms_file()** described in 8.5.4 above. It is merely ensuring that all the steps detailed in the instruction file make one of more coherent SMS messages, before processing by **process_bulksms_file()**.

To understand the workings of **check_file_content()**, one should, instead, understand **process_bulksms_file()** as detailed in 8.5.4 above.

Note that it is only the format of the Bulk-SMS instruction file that is checked. The validity of the phone-number is not itself checked.

10.5.6 OTHER FUNCTIONS

10.5.6.1 **process_bulksms_msg**

process_bulksms_msg() actually send the already-processed SMS message to one of the four priority Bulk-SMS queues. First it optionally finds the international prefix (e.g. 064), if any, by using **get_dest_frm_country_code()** to look up the mdsdestm database table. Next, **process_bulksms_msg()** checks that the target priority queue is operational, before using the <System_Name> standard **send_message()** function to post the message to the queue.

10.5.6.2 **check_unique_msg**

check_unique_msg() function parses the complete SMS message text appearing after a UNIQUEID tag.

10.5.6.3 **extractBatchNoAndPin**

Email messages containing Bulk-SMS instruction files are sent to addresses such as:

000000000000000989898-1234@bulksms.<Company_Name>.com.au

where:

000000000000000989898 is the ‘Batch Number’, to do with the authority to send SMS messages arranged between <Company_Name> and the A-Party client
1234 is the PIN number assigned to this A-Party client

extractBatchNoAndPin() parses the ‘To:’ email address (‘000000000000000989898-1234’) in order to extract the Batch Number and PIN number.

10.6 STOPPING MDSEM03

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)
3. If, when the priority queue is checked, the status comes back as ‘EXIT’, then mdsem03 will exit.

11 mdssms01

11.1 MDSSMS01 OVERVIEW

mdssms01 is the <System_Name> application that:

1. extracts individual Bulk SMS messages from each of the four priority queues
2. posts them to the bulk queue.

Doing little checking and processing, mdssms01 is a reasonably simple program.

11.2 RUNNING MDSSMS01

mdssms01 is executed from each site's start.mds <System_Name> startup script.

11.2.1 COMMAND LINE PARAMETERS

Parameter	Description
<u>Mandatory</u>	
-t <port>	The port (e.g. /dev/dum77) to assign to this instance of mdssms01
<u>Optional</u>	
-n <number>	Don't post to the bulk queue if it is longer than < number > messages
-p <number>	Number of priority queue messages processed per cycle
-s <seconds>	How long to sleep between queue-reading cycles of reading the queues.
-D	run in debug mode, listing detailed run-time information.

Table 76: mdssms01: Command Line Parameters

11.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute mdssms01 at one <System_Name> site (Perth) is:

```
$MDS_BIN_DIR/mdssms01 -t /dev/dum77 -n 30 -p 5 2>$LOG_SCR
```

indicating that mdssms01 is to run:

- on port /dev/dum77
- processing up to 5 messages from the priority queues each (1 second) cycle.
- sleeping if more than 30 messages are in the bulk queue
- stderr (file descriptor 2) being copied to LOG_SCR, defined in start.env to be / dev / tty1p0

11.3 PROGRAM DESCRIPTION

As described above, mdssms01, running as a background daemon, reads messages from the four Bulk SMS priority queues and posts up 5 (let's say) of these messages to the bulk queue each second.

In mdssms01's `main()` function, after checking the bulk queue, `process_pqu_recs()` is called to process any, and possibly all, of the four priority queue messages. `process_pqu_recs()`, using `read_pqu_rec()`, extracts the messages from priority queue #1 and posts them to the Bulk Queue with `queue_write_rec()`. Provided mdssms01 hasn't exceeded its messages-posted-per-cycle limit, queue #2 is now processed in the same way, then #3, then #4. Processing continues until either all the messages in all the queues have been sent, or the messages-posted-per-cycle limit has been reached. Once this traffic limit is reached all processing stops, mdssms01 sleeps for one (or more) seconds, and processing begins again at queue #1. Thus no messages in queue #3 (for example) will be processed until both queue #1 and #2 are empty.

After posting an SMS message, `update_pqu_header()` is called to update the appropriate priority SMS queue.

11.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into mdssms01.

11.4.1 SOURCES FILES

Source File	Description
alarmrtn.c	Generic alarm routine.
gensubs.c	Contains various general purpose routines that are separately compiled.
globvar.c	Functions to access the Global Variable shared memory used by multiple programs.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
logerr.c	Error logging functions
mdscdr.c	Contains various general purpose routines that are separately compiled.
mdsemqmain.c	Generic message email queuing routine.
mdsginit.c	A number of functions which perform commonly used initialisation code segments.
mdshisio.c	Generic history file handling routines.
mdskbld.c	Contains functions to build and unbuild pager keys, and setup a dummy pager.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqrtn.c	Generic message queuing routine.
mdsseck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssetup.c	This module contains routines for a program to obtain setup definitions from a text setup file.
mdssms01.c	Distribute Email Priority Queues data to Bulkq
mdsstamp.c	Date time stamp & sequence number generation
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions which perform operations to do with time.

Table 77: mdssms01: Source Files

11.4.2 HEADER FILES

Header File	Description
gentypes.h	User defined types and 'booleans': user defined types
gettime.h	C time field definitions
globvar.h	Global variable shared mem defs, memory structures
macros.h	User defined macros.: definitions & macros for common fns
mdsaprec.h	A-Party master file: Email A-Party file
mdscdr.h	CDR definition files
mdscdrec.h	CDR data structure
mdsctype.h	Used for validating message chars.
mdsdlrec.h	Networking Delay monitoring file
mdsflsz.h	Standard <System_Name> includes.: field length & definitions used by MDS
mdshirec.h	History primary recs file: New History logging file
mdshsrec.h	Secondary History file (audits): History secondary recs file
mdslirec.h	History primary file
mdsparam.h	Parameter values used by PMS.
mdsparec.h	Pager file - keep mdskbld.c happy: (PAG) Pager Master File
mdsporec.h	only for security levels: Ports security master file: Ports Record
mdsquec.h	Queue header file

Table 78: mdssms01: Header Files

11.4.3 DATABASE TABLE

Each of these header files contain (only) a C-struct that matched the structure of a database table used by mdssms01.

Database Table	Header File	Description
hiccyymmdd	mdshirec.h	the 'logical' history
hsyymmdd	mdshsrec.h	History secondary storage
mdsacrec	mdscdrec.h	Alt call details master
mdsaparty	mdsaprec.h	MDS A-Party
mdscdrec	mdscdrec.h	Call details master
mdsdelay	mdsdlrec.h	Delay monitoring
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsqurec.h	Queue header file

Table 79: mdssms01: Database Tables

11.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment variable	Description
MDS_BIN_DIR	All the <System_Name> executables
MDS_DATA_DIR	The root of the data directories, queues, database and history.
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide B Brisbane P Perth C Canberra

Table 80: mdssms01: Environment Variables

11.5 MDSSMS01: MAJOR FUNCTIONS

This section lists the most important functions in mdssms01. As it is a simple program there aren't many functions.

11.5.1 FUNCTION: MAIN

mdssms01, being simpler than the other email processing programs, has its program logic written to its `main()` function. After calling `initialise()`, an infinite loop is entered to process the priority queue emails. First the size of the bulk queue is checked, following which `process_pqu_rec()` is used to transfer emails from the priority to the bulk queue.

Upon return from `process_pqu_rec()`, the number of emails processed is compared to the emails-per-cycle limit.

- If the limit is not exceeded (all emails have been processed): sleep for 20 seconds.
- If this limit is exceeded (more emails are probably waiting): sleep for 1 second – or whatever is passed as the ‘-s’ command-line parameter.

The sleeping is achieved by calling the <System_Name> `sem_check()` function, which, via a third function, actually puts the process to sleep.

11.5.2 FUNCTION: INITIALISE

The `initialise()` function sets up mdssms01's run-time environment, via the following steps:

- Reads the environment variables, as per 9.4.4
- Parse the command-line. As per 9.2.1
- Put the process in the background

the `background()` function is called to put mdssms01 into the background. This short function, coded in mdsginit.c, simply forks the process and kills the parent, leaving the child running in the background.

- Signals from Ctrl-C, alarms, and terminations (kill commands) are assigned to appropriate handlers
- Set shut-down semaphore

- Open the mdspager Pager Master database table, the bulk queue (which is probably already open) and connect to the priority queues.

11.5.3 FUNCTION: PROCESS_PQU_RECS

As described, **process_pqu_rec()** cycles through the four priority queues – from Priority Queue #1 to Priority Queue #4 – until either the limit number of emails per cycle is reached²², or all the queues have been processed.

That's pretty much it.

In a loop, over the priority queues, **process_pqu_rec()** uses **read_pqu_rec()** to read the next email message from the current priority queue, and **queue_qrite_rec()** to write it to the bulk queue.

11.5.4 FUNCTION: READ_PQU_REC

read_pqu_rec() is basically a wrapper for **QU_queue_read_msg()**, the general <System_Name> queue reading function. After reading the record, it updates the priority queue's header record – as it should. After a message is written (by **queue_write_rec()**), **update_pqu_header()** also updates the queue header record. The queue header record keeps track of the state of the queue.

11.5.5 FUNCTION: QUEUE_WRITE_REC

queue_write_rec(), coded in mdsqmain.c, writes a message to the bulk queue

11.5.6 FUNCTION: UPDATE_PQU_HEADER

update_pqu_header() maintains the queue's nextsend and nbr_sent (how many messages have been sent) fields. It basically wraps a call to the <System_Name> standard **QU_queue_update_hdr()** function.

11.6 STOPPING MDSSMS01

1. Set the shutdown_semid to non-zero.
4. Find its PID and kill -9 it (as root)
5. If, when checking the Bulk Queue, the status returned is 'EXIT', then mdssms01 will terminate.

²² 5 per second now. Probably 500 per second in the not-too-distant future.

12 skymds96a

12.1 SKYMD96A OVERVIEW

skymds96a is the ‘<System_Name> II’-to-<System_Name> interface, reading messages from <System_Name> II’s message queue²³ – an Oracle database table – into <System_Name>’s bulk queue. This ensures that all <System_Name> II messages are delivered normally via the <System_Name> distribution systems.

<System_Name> II is the bionic web-based wrapper for <System_Name>

12.2 RUNNING SKYMD96A

skymds96a is run from each <System_Name> server’s start.mds start-up script, start.mds being found at MDS_BIN_DIR: usually `/opt/mds/bin/start.mds`

12.2.1 COMMAND LINE PARAMETERS

Parameter	Description
<u>Mandatory</u>	
<code>-t <comms port></code>	The communication port (e.g. /dev/dum99) that skymds96a is registered with
<u>Optional</u>	
<code>-c</code>	don’t check SMS security.
<code>-D</code>	run in debug mode, listing detailed run-time information.
<code>-n <Oracle Server></code>	One or more Oracle servers, separated by commas. See the example below.
<code>-?</code>	Print simple usage instructions, similar to this table.

Table 81: skymds96a: Command-line parameters

12.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute skymds96a at one <System_Name> site is:

```
export ORACLE_HOME=/opt/ora/product/8.1.6;
$MDS_BIN_DIR/skymds96a -n SSNSW,SSVIC -t /dev/dum67 2>$LOG_SCR
```

indicating that skymds96a is run:

- connecting to the (Oracle, web-based) <System_Name> II servers ‘SSNSW’ and ‘SSVIC’.
- registered to communication port ‘/dev/dum67’
- the stderr stream (file descriptor 2) is copied to LOG_SCR, which is defined in start.env to be /dev/tty1p0

As just noted, MDS_BIN_DIR is usually set to /opt/mds/bin by the start.env environment settings file.

12.3 PROGRAM DESCRIPTION

skymds96a is the <System_Name> II (Web-based) to <System_Name> interface, reading messages from <System_Name> II’s message queue – an Oracle database table – to <System_Name>’s bulk queue. The `main()` function, after calling `initialise()` to set up its environment, passes control to `process_request()` to implement the <System_Name> II to <System_Name> connection.

`process_request()` sets up an infinite loop in which the following tasks are repeatedly performed. First connections are checked and optionally made to the primary server (the local <System_Name> II server, listed first on the command line and always connected) and the secondary server (a non-local <System_Name> II server, listed second on the command line and only processed if it is not otherwise being served). After checking its connections, `process_request()` calls `get_next_select_msg()` to extract the next message from the Oracle database,

²³ <Company_Name>Net II’s message queue has been named, wait for it ... the Bulk Queue. In order to avoid any more confusion I will not refer to their (<Company_Name>Net II’s) message queue by that overloaded name.

and `REQ_id_page()` to format the retrieved message into a <System_Name> bulk queue record. Finally `bulkq_db_update()` is used to post the bulk queue record to <System_Name>'s bulk queue.

12.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds96a.

12.4.1 SOURCES FILES

Source File	Description
skymds96a.c	Process page messages stored in the <System_Name> II Oracle DB.
alarmrtn.c	Generic alarm routine.
gensubs.c	Contains various general purpose routines that are separately compiled.
iserror.c	A function to turn a C error message into a human-readable string
isstart.c	Wrapper for C-ISAM isstart function – choose index and position the read-point
mdscdr.c	Contains various general purpose routines that are separately compiled.
mdsemqmain.c	Generic message email queuing routine.
mdsextco.c	Allow External User Interface (PET/MERP) to initiate enhanced services belonging to indials
mdsginit.c	A number of functions which perform commonly used initialisation code segments.
mdskbld.c	Contains functions to build and unbuild pager keys, and setup a dummy pager.
mdslog.c	Generic error logging.
mdsoracle.c	Functions to connect <System_Name> to an Oracle database, as in <System_Name> II.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurtn.c	Generic message queuing routine.
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions which perform operations to do with time.
security.c	Various Security Routines.
tnppcrc.c	Calculate the CRC for a given TNPP string.

Table 82: skymds96a: Source Files

12.4.2 HEADER FILES

Header File	Description
gentypes.h	User defined types and 'booleans': user defined types
gettime.h	C time field definitions
macros.h	User defined macros.: definitions & macros for common fns
mdsaprec.h	Email A party file
mdscdr.h	CDR definition files
mdscdrec.h	CDR data structure.
mdsctype.h	Used for validating message chars.
mdsextco.h	Header file used for the External Contacts
mdsflsz.h	Standard <System_Name> includes.: field length & definitions used by MDS
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsmtip.h	Short Message Transaction Protocol
mdsparam.h	Parameter values used by PMS.
mdsparec.h	Pager Master File
mdsporec.h	Ports security master file
mdspsec.h	Password master file
mdsqrec.h	Queue master file.
mdsserec.h	Security Group master file: Security Groups master file
security.h	Structure and defines for security groups

Table 83: skymds96a: Header Files

12.4.3 DATABASE TABLES

The following lists all of the database tables used by skymds96a. The header files contain (only) a C-struct that matches the structure of a database table used by skymds96a.

Database Table	Header File	Description
mdsaparty	mdsaprec.h	A-Party customer details.
mdsacrec	mdscdrec.h	Alt call details master
mdscdrec	mdscdrec.h	Call details master
mdsline	mdsline.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdspassw	mdpsrec.h	Password username
mdsqxx.txt	mdsquec.h	Destination queues
mdssecur	mdsserec.h	Security codes

Table 84: skymds96a: Database Tables used

12.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description
MDS_BIN_DIR	All the <System_Name> executables
MDS_DATA_DIR	The root of the data directories, queues, database and history.
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide B Brisbane P Perth C Canberra

Table 85: skymds96a: Environment Variables

12.5 SKYMD96A: MAJOR FUNCTIONS

NB: if reading the code, note that there are functions and many stretches of code in skymds96a.c no longer called. skymds96a is derived from an earlier ‘skymds96’, and much superceded code has remained in the skymds96a.c source file. A list of “Functions No Longer Used” is included at the end of this section.

12.5.1 FUNCTION: MAIN

The simple **main()** function calls **initialise()** to set up skymds96a’s program ‘environment’. After doing this it passes control to **process_request()** to undertake the main logic of the program.

12.5.2 FUNCTION: INITIALISE

The **initialise()** function sets up skymds96a’s run-time environment, via the following steps:

- Reads in the appropriate Environment Variables – see 10.4.4 above
- Parse the command-line. As per 10.2.1 above. If the port isn’t supplied (no ‘-t’ option), or a ‘?’ is passed, the **usage()** function is called to write a short description of command-line details.
- Put the process in the background

the **background()** function is called to put skymds96a into the background. This short function, coded in mdsginit.c, simply forks the process and kills the parent, leaving the child running in the background.

- Set Signal handlers

Handlers for a large number of signals are set. Timeouts, debug signals and termination signals are specifically handled, a long list of other signals are directed to the **`sig_dump()`** function, which merely writes an information line to the MDS_ERR log file.

- Record the port, the shut-down semaphore, and the local Machine ID
- Open a file handle to the <System_Name> bulk queue
- Open database tables. See 10.4.3 above.
- Check that there is a valid record in the line database table for the port that was supplied
- use **`validate_user()`** to ensure that the current_operator is a permissible user of skymds96a.

12.5.3 FUNCTION: PROCESS_REQUESTS

`process_requests()` is the central process of skymds96a, managing the transfer of messages from the <System_Name> II message queue to <System_Name>'s bulk queue. Control passes to it upon start-up, and does not leave until the program is terminated.

- connect a <System_Name> II Oracle database message queue

Generally skymds96a is configured to read from two separate <System_Name> II message queues, the 'local' <System_Name> II queue, in the same city as the <System_Name> system; and a non-local <System_Name> II queue, as a back-up for that server. As the state of both the local and the non-local queues may change at any time, the need – and ability – to service each of these queues is checked constantly – before each and every <System_Name> II to <System_Name> message transferral.

skymds96a may actually service any number of <System_Name> II message queues. On each iteration through **`process_requests()`**, the next server in its list of servers is, if necessary, serviced; an arbitrary large number of queues being thus being supported. The first <System_Name> II server supplied on the command-line is always processed, ideally the local <System_Name> II message queue. Any other skymds96a program (running on another <System_Name> system) that may have been managing this <System_Name> II message queue will now find that the queue is already being handled, and stop processing it. Similarly, each other, non-local, server will be processed by this local skymds96a program if and only if it is not currently being processed by its own, local, skymds96a program (i.e. as a back-up).

- process all the messages in the <System_Name> II's message queue.
 - ~ **`get_next_select_msg()`** is called to extract a message from the <System_Name> II queue,
 - ~ **`REQ_id_page()`**:
 1. converts the <System_Name> II message into a <System_Name> bulk queue format
 2. calls **`queue_main()`** to post the record to the <System_Name> Bulk Queue
 - ~ **`bulkq_db_update()`** updates the <System_Name> II message queue²⁴.

These calls are looped until the number of messages present in the <System_Name> II message queue when the process started are successfully transferred. Tests that any non-local <System_Name> II servers still need to be supported are also made during each of these read → process → post loops.

²⁴ See footnote #1. The <Company_Name>Net II message queue is also named "Bulk Queue", thus the name of this Function: "bulkq_db_update()"

12.5.4 FUNCTION: REQ_ID_PAGE

REQ_id_page() formats a <System_Name> II message queue record for insertion into the <System_Name> bulk queue.

- **check_pager_and_security()** ensures that the destination pager number is valid.
- An attached phone-number indicates that the <System_Name> II record is an SMS, not a pager message. If a phone number is found to be attached to the <System_Name> II record, **check_security_level()** (not the **check_security()** described below) is used to ensure that the sender of the message is allowed to send SMS messages.
- Next, checks are done to ensure that if advanced services – currently only Escalations and Rosters²⁵ – have been requested for this customer, they have been set up and are currently available. If not, **REQ_id_page()** is exited.
- The message itself is validated, ensuring that there are no illegal or unprintable characters in the message, and trailing blanks are stripped.
- **setup_queue_rec()** prepares a message record for the <System_Name> bulk queue
- **queue_main()** posts the message record to the bulk_queue.

12.5.5 CHECK_PAGER_AND_SECURITY

check_pager_and_security() is basically a wrapper for checking the mdspager table

- the pager number exists,
- the pager number is active,
- if an SMS is being sent, permission exists for sending SMS messages
- call **check_security()** (in security.c) to test the user's security – password and ID

12.5.6 OTHER FUNCTIONS

12.5.6.1 get_next_select_msg

An external function, coded in mdsoracle.c, that reads a record from the <System_Name> II message queue – an Oracle database table. A simple function that wraps an SQL call on the Oracle database.

12.5.6.2 bulkq_db_update

Updates the Oracle message queue to indicate that the record just read has been successfully processed.

12.5.6.3 check_security

Test the user's login-ID and password are OK. Uses the functions in (security.c) **check_security_level()** and **validate_oper_subscriber()** to do this, both of which, pretty much, just look up mdspassw and mdssecur, the password and security code tables.

12.5.6.4 validate_message

Is the message too long? - truncate it, does it have non-ASCII characters? - convert to spaces, and (optionally) send an error message if either of these imperfections have been found.

12.5.6.5 ss_server_served

Test to see if a particular <System_Name> II message queue is being processed by any instance of skymds96a. This is used in **process_requests()**, when that function is testing to see if it should deliver the messages in any as-yet unserved <System_Name> II message queue.

12.5.6.6 setup_queue_rec

Quite a long function, but basically just formatting a <System_Name> bulk queue record with empty, default, and / or appropriate values for delivery to said bulk queue.

²⁵ Described in Volume 1, section 4.1.1 and in Volume 1's glossary (section 10).

12.5.6.7 validate_user

validate_user() is used to ensure that the user of skymds96a is, in fact, a valid user.

In the current version of skymds96a (mid 2002), the user is read from (and fixed) in the mdsports table. The user happens to be SSVIC (say), the name of the local <System_Name> II database that is expected to be processed by the instance of skymds96a associated with the port that is found in the same record as SSVIC (say). Thus, a set of rights are associated with the database. In other programs, the operator is just that, the user – the call centre operator or the customer using the system²⁶.

12.5.7 FUNCTIONS NO LONGER USED

As noted above, a number of functions appear in the skymds96a source code that are no longer used. For anyone who actually is going to look at the code, ignore the following functions:

- `get_next_trans()`
- `REQ_polling()`
- `REQ_cap_page()`
- `vaildate_cap_details()`
- `REQ_line_check()`
- `REQ_test()`
- `REQ_invalid_trans()`
- `build_generic_packet()`
- `build_line_rep_packet()`
- `build_test_packet()`
- `build_err_packet()`
- `process_reply_packet()`
- `open_port()`
- `open_serial_port()`
- `connect_to_lan`
- `send_error_tran()`
- `send_alarm()`

The last third of the `process_requests()` function, inside the `#ifdef never` blocks, has also been removed.

12.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

12.7 STOPPING SKYMD96A

1. Set the shutdown_semid to non-zero.
2. Find its PID and `kill -9` it (as root)
3. Any termination signal (any kill) is caught, `termination_requested` is set, and skymds96a quits at the start of the next `process_requests()` loop

²⁶ In the case of direct connections to the <Company_Name>Net system via PET or MERP

13 skymds02

13.1 SKYMDS02 OVERVIEW

skymds02, implementing the P.E.T. protocol, runs as the P.E.T. receiver for the <System_Name> system. P.E.T. (“Pager Entry Terminal”) is a standard protocol for interfacing to paging systems and is used by most customers when connecting their computers to the <System_Name> systems. An alternate protocol is MERP – <Company_Name>’s proprietary interface system. MERP, very similar to PET, has some extended functionality, such as the ability to retrieve historic messages. (See the skymds70 documentation for details of the MERP implementation.)

13.2 RUNNING SKYMDS02

skymds02 is executed from the <System_Name> start-up script “start.mds”, with a number of individually configured instances running concurrently, each having been initialised with different ports and (possibly) other command-line options. At the time of writing between 8 and 27 instances of skymds run at any one time on the various Australian <System_Name> systems. Once started, skymds02 programs run for months without needing to be restarted.

13.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-t <comms port >	communication port
-D	debug mode, listing detailed run-time information.
-s <seconds >	the general “sleep time” – How long to wait between checks for another log-in
-S	the same as -s
-I	Internet usage
-r <type>	allow restricted pagers of given type
-c	check SMS security for sms pagers
-l	loose password mode
-n	no hangup mode
-m <number >	maximum number message allow to send
-e < number >	maximum number of consecutive errors
?	just print the usage instructions on the stdout

Table 86: skymds02: Command-line Parameters

Only a valid port – can be opened and locked – is necessary on the command-line. In this skymds02 is different from many other <System_Name> programs in that skymds02 uses the port to receive its data-stream from the user (via a modem connected to the port) while, in many other applications the port itself is not used, only being compared to a database port record as a sort of unique ‘tag’, ensuring that the ‘tag’ a port identifier appears in the ports database table and is being referenced only once. In these programs the port(s) is/are not actually used – in skymds02 it is used.

If an invalid, or no, port is supplied, or a ‘?’ parameter is passed, usage instructions are printed to stderr and the program exits.

13.3 PROGRAM DESCRIPTION

As described above, skymds02 is a daemon that collects, converts, and transfers PET protocol messages from client modem connections to the <System_Name> Bulk Queue for distribution to appropriate SMS, pager, or other destinations.

After setting itself up via `initialise()`, control passes to a loop (within `await_login()`) which polls the communications port for either <ESC> or <CR>, indicating a logon request. When one arrives the message type, and the “operator” are also extracted. Main then calls `get_valid_message()` to actually gather and pre-process the message, the message itself being stored in the global PET Transmission record “pe”. Observing a pause (if the command-line “-s” parameter has been used) between consecutive messages from the same user, the message is inserted onto the Bulk Queue. The remainder of the `main()` loop is concerned with testing various error and termination conditions, failing which the loop goes around again, to wait for the next message.

13.4 PROGRAM STRUCTURE

13.4.1 SOURCE FILES

The following are the source files that are built into a skymds02 executable.

Source File	Description
skymds02.c	Pet protocol receiver.
gensubs.c	Contains various general purpose routines that are separately compiled.
iserror.c	Single simple function that converts an error number to an error string. Customised strerror().
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
maskpagr.c	Read pager file with line file driven masking.
mdscdr.c	Contains various general purpose routines that are separately compiled.
mdsemqmain.c	Generic message email queuing routine.
mdsextco.c	Functions to allow PET and MERP to initiate enhanced services belonging to indials
mdsginit.c	A number of functions which perform commonly used initialisation code segments.
mdskbld.c	Contains functions to build and unbuild pager keys, and setup a dummy pager.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurt.n.c	Generic message queuing routine.
mdssetup.c	This module contains routines for a program to obtain setup definitions from a text setup file.
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions which perform operations to do with time.
security.c	Various Security Routines.

Table 87: skymds02: Source Files

13.4.2 HEADER FILES

The following is a list of the *.h files used in skymds02

Header File	Description
gentypes.h	User defined types and 'booleans': user defined types
gettime.h	C time field definitions.
macros.h	User defined macros.: definitions & macros for common functions
mdsaprec.h	A-Party customer details
mdscdr.h	CDR definition files
mdscdrec.h	Alternative call details master
mdscdrec.h	Call details master
mdsctype.h	Used for validating message chars.
mdsextco.h	Header file used for the External Contacts
mdsfldsz.h	Standard <System_Name> includes.: field length & definitions used by MDS
mdslirec.h	Indial number details.
mdsparam.h	parameters used by PMS system: Parameter values used by PMS.
mdsparec.h	Pager master
mdsperec.h	PET trans. record layout - not a Database Table
mdsporec.h	Ports security
mdspsec.h	Password username
mdsquec.h	Queue record. This is the distribution queue header, but used for the Bulk Q
mdsserrec.h	Security codes
security.h	security functions: Structure and defines for security groups

Table 88: skymds02: Header Files

13.4.3 DATABASE TABLES

Database Table	Header File	Description
mdsaparty	mdsaprec.h	A-Party customer details
mdsacrec	mdscdrec.h	Alt call details master
mdscdrec	mdscdrec.h	Call details master
mdsline	mdslirec.h	Indial number details.
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdspassw	mdspsec.h	Password username
mdssecur	mdssrec.h	Security codes
mdsqxx.txt	mdsquarec.h mdsperec.h	Queue record for the distribution queue, but used here in the bulk queue PET trans. record layout - not a Database Table

Table 89: skymds02: Database Tables

13.4.4 ENVIRONMENT VARIABLES REFERENCED

Environment Variable	Description
MDS_MACHINE_ID	What machine (state) are we on/in e.g. M=Melbourne,
MDS_DATA_DIR	Where all the <System_Name> database tables and queues are, generally /var/mds/data
Program Specific Environment Variables	
MDS_PET_INTER_CHAR_TOUT	Inter-character timeout (“180” = 3 minutes)
MDS_PET_HANGUP_TOUT	General Timeout (“180” = 3 minutes)
MDS_PET_PAG? ACTIV_TOUT	An activity timeout based on the PET message type

Table 90: skymds02: Environment Variables

The ‘Program Specific Environment Variables’ are setup by the start.mds for a sub-group of programs when they are run.

13.5 SKYMD02: MAJOR FUNCTIONS

13.5.1 FUNCTION: MAIN

The skymds02’s **main()** function contains and controls the behaviour of the program, calling most other important functions directly. First it calls **initialise()**, before entering a pair of nested infinite loops, the outer one waiting for a login by some external process, the inner one processing all the PET format messages sent from the current logon. The external loop calls **await_logon()**, which, as the name suggests, does not return until a logon has been received. Once it has, the inner loop goes through the following steps:

- Gets a message

get_valid_message() reads a message from the PET connection, into the global PET message transmission record (‘PEREC’) structure.

- Test the message count for this logon

As there is a maximum number of messages that can be sent per logon, the count is tested against this maximum. Also, if consecutive messages are being sent to the same pager number, skymds02 will sleep for a short pause.

- The message is added to the Bulk Queue

queue_main() is called to add the message to the bulk queue.

- A few house-keeping duties

Print a debug message, log any “loose user” messages, copy the time to shared memory (so that lack of activity can be monitored), and test for port errors. Loose users are those that have been allowed to login with-out testing security. Possibly for maintenance, testing or as part of a special third-party arrangement.

- Test for termination of the connection
 - ~ Three consecutive errors and the user is logged out.
 - ~ Test for time-outs
 - ~ Test for log-outs
- Loop

If the connection is terminated, break from the inner loop (processing messages for this login) and return to the top of the outer loop, waiting for the next valid logon. Else, stay in the inner loop, again calling `get_valid_message()` for the next message

13.5.2 FUNCTION: INITIALISE

`Initialise()` follows the same behaviour as most other <System_Name> initialise functions.

- Read in the relevant environment variables
- Parse the command line
- Put the process in the background
- Set up handlers for a range of signals
- call `open_port()` to open and lock the communications port
- set up the shared memory in which to store timestamps – for inaction to be detected
- Open all the files needed
- Connect to the security shared memory block
- Check that the communications port supplied on the command-line matches one in the Indial (“Line”) database table

13.5.3 FUNCTION: AWAIT_LOGIN

`await_login()`, once called, doesn’t return until a valid logon has been established. It has two parts.

During its first part, `await_logon()` reads the communications port until an <ESC> character (ASCII code 33) is received, signifying the beginning of a message. It also test for various other conditions:

Condition	Response
5000 polls of the port with no message	re-start the modem
EOT “End of Transmission” (ASCII code 04) received	(1) Raise an alarm (2) re-start the modem
5000 inlaid characters received	re-start the modem
5 times through: none of the above, and still no message	re-start the modem

Table 91: skymds02: Handling Erroneous PET Transmissions

During the second part, we know we have got something that seems like a valid logon. Now to test it’s OK.

- logon string

Test various attributes of the logon string submitted, then pass it to `validate_user()` to test the permissions of the user. If the password is incorrect, force the modem to re-start and return to waiting for the next <ESC> (start of login) character.

- what type of PET message

The first character indicates what type of PET message this is. After matching this value (1,2,3,4, or 5) to a string of the form MDS_PET_PAG?_ACTIV_TOUT, the matched string is searched for in the environment to get its value, which is assigned to a time-out value for this type of PET messages. Values typically supplied are 25 (25 seconds), 2880 (48 minutes) or 86,400 (1 day).

13.5.4 FUNCTION: GET_VALID_MESSAGE

After `await_login()` supplies a valid PET connection, `get_valid_message()` is called between one and ‘max_message_sent’ times to copy each message to the global PET message transmission record. After each `get_valid_message()` call `queue_main()` is used to copy the message retrieved to the Bulk Queue.

`get_valid_message()` accomplished the following tasks:

- Acknowledges the previous message

Sending the ACK character here inserts the time of the inter-message sleeping between the message processing and acknowledgement, which ensures that messages aren’t lost by over-hasty acknowledgement

- Prepare to receive a message

In a loop, characters are read until a STX (“Start of Text” ASCII value 2) or EOT (“End of Transmission” ASCII value 4) value is read. Within this loop, the first thing is checking that the program is not being halted. After this a few errors and inconsistencies (time-outs, attempted re-logons, 256 consecutive error characters) are checked and the loop restarts on all non-STX and non-EOT characters. If an EOT is found, the `get_valid_message()` function is exited. If STX is found, the message is processed.

- Receive the message

1. First, read the pager number from the buffer. Some error checking and processing of the number is also done. Next, read the message proper.

2. Establish what city the message is being sent to. Different cities mean that the message will have to be routed to different networks.
3. Read the check-sum number.

All done.

- Check the message

1. Ensure that the check-sum number is valid.
2. Are all the security / access requirements satisfied?
3. If the message record indicates that advanced services are required; are they properly set up?
4. Check that there is a valid destination city attached to the message.

if any of these fail, exit the function, which effectively abandons the message.

5. Truncate the message if it is too long. The maximum length is stored, for every pager, in the mdspager table.
6. Replace any invalid characters with spaces. If there are any, send the message, but return a RS3 flag (not ACK)

13.5.5 QUEUE_MAIN

`queue_main()`, coded in mdsqmain.c and compiled into a separate module (mdsqmain.o), is called to add the message to the Bulk Queue.

13.5.6 FUNCTION: STAT_OPERS

skymds02 groups its statistical reporting tasks in one the one Function: **stat_ops()**. These tasks are of two types.

1. Note that an event has happened: Initialisation, logon, logoff
2. Respond to an event, and write all collected statistic to a log file. The list of log messages written appears in the Log Messages section. ‘Enquiry’, ‘Date_change’, and ‘Finalise’ are examples of this.

For all actions except ‘Enquiry’, the log messages are written to a file named: MDS_DAT_DIR/logs/mdspet???.txt, e.g. /var/mds/data/logs/mdspet24.txt. Enquiries are written to stderr.

13.6 LOG MESSAGES PRODUCED

The log messages – those that aren’t just sent to stderr – are sent to the directory specified by the environment variable “MDS_DATA_DIR” (typically “/var/mds/data”) with “logs” appended (i.e. “var/mds/data/logs”) and stored in the file mdspet???.txt where?? the 8th and 9th characters from the port-location on which skymds02 is running. This value is passed to skymds02 as a command-line option, and must be in the form “/dev/mlb12”.

Note that skymds02 has an additional logging process, as described below in the stat_ops section.

13.6.1 DEBUG-MODE ONLY: LOG MESSAGES WRITTEN TO STDERR

Consecutive messages are being sent to the same destination – sleep. Else can create traffic jams in the system
Sleeping for <sleep_time> seconds

After queue_main returns, flag that it was successful
MESSAGE SUCCESSFULLY SENT

Flag each logon request in the main() loop
LOGON REQUEST<timestr>

Timeout in main() loop
TIMEOUT<timestr>

Exiting the PET connection
END_OF_SESSION<timestr>

Sleeping after a disconnect of a PET session
Sleeping for <sleep_time> seconds

In the await_logon() function
wait_logon()

To track the time between receiving consecutive characters in await_logon()
<timestr>

Read an end-of-line character while in await_logon()
03 read logon <cr> errno=> tmsfr=>

Other invalid character read
04 logon char!=<cr> is=<>invalid_char_cnt=

couldn’t write 4 ID bytes to the comms port
05 write lun_comms tmsfr!= 4

invalid logon string in await_logon()
06 invalid logon string

The operator – the external PET user – connected.
operator <current_operator> log on success

Security failure: The pager ID not in the database

33 **pager not on file**

Security failure: Pager marked as inactive

34 **pager out of service**

Security failure: SMS (despite message) not found

33 **pager not on file**

check_security() found.

35 **illegal pager for this operator**

The date has been updated

45 **date changed**

Couldn't write the disconnect sequence to the port

07 **write disconnect sequ errno = <errno >**

In the get_valid_message() function

get_valid_message()

To track the time between receiving consecutive characters in get_valid_message()

<timestr >

A timeout between characters in get_valid_message()

09 **activity timeout**

Log that the ‘number’-th STX signal received

12 **read <STX> errno=<errno > cnt =< number >**

The ‘number-th’ non STX and EOT character received, while waiting for STX

13 **non<STX>/<EOT>recv, char=< c >, cnt_stx_err = < number >**

Timeout waiting for a end-of-line after an EOT (End of Transmission)

14 **time out on <CR> after <EOT>**

Read a non-character after an EOT (End of Transmission)

15 **read <CR> after <STX> errno=<errno >**

Timeout reading the pager number from the port

16 **read <pager number> timeout**

No End-of-line at the end of the pager number

19 **read <pager number> no <CR>**

Timeout reading message

20 **read <message> timeout**

No End-of-line at the end of the message

23 **read <message> no <CR>**

Timeout reading the destination city

24 **read <city> timeout**

City-code incorrect length – (Bug: this message will always be emitted)

26 **read <city> invalid length**

No End-of-line after the city-code

27 **read <city> no <CR>**

Timeout reading the ‘end-of-text’ and the ‘check-sum’.

28 **read <ETX><checksum> timeout**

Read the ‘end-of-text’ and the ‘check-sum’

30 **read <ETX><checksum> length**

No End-of-Text character in the message

32 **read <ETX><checksum> no <ETX>**

No end-offline after ‘end-of-text’ and ‘check-sum’

31 **read <ETX><checksum> no <CR>**

Check-sums don’t match

Sent checksum=>.*sCalculated checksum=>.*s

Can’t provide the services requested for this message

46 **Enhanced Service Funct Setup Incomplete**

Invalid city code

38 **invalid dest for sms=<city>.**

Invalid city code

38 **invalid dest - not in dest file**

Message truncated

39 **message too long - truncated**

Non-printable and/or non-text characters in message

40 **illegal characters in the message**

Hanging up the modem

hangup()

Closing the communication port

close() comms port

Opening the communications port

open() comms port <our_tty_str> ok

A timestamp written when an ACK signal sent

<timestr>

a NAK – negative acknowledgement sent

<timestr>

a RS sent

<timestr>

Program has started

<date-time> SKYMD502 begun on <comms port>

Someone has logged on

<date-time> <operator> logged on <comms port>

Someone has logged on

<date-time> <operator> logged off <comms port>

13.6.2 LOG MESSAGES WRITTEN TO .../LOGS/MDSPET??.TXT AND STDERR

These are the log-messages produced by the stats_opers function, written in the form:

```
-----  
STATS SINCE < date – time >          (Header)  
:  
23    read <city> invalid length      (count)      (log string)  
24    write sysoutput status  
25    :  
:  
< Date time > text string  
-----
```

to both stderr and to the file /var/mds/data/logs/mdspet??.txt. As the examples show, the output is a descriptive title followed by a count of the number of occurrences of that (usually) error.

And the messages are:

log on	read <pager number> timeout	read <ETX><checksum> no <CR>
log off	read <pager number status>	read <ETX><checksum> no <ETX>
message received	read <pager number> wrong length	pager not on file
read logon <cr> status	read <pager number> no <CR>	pager out of service
logon char!= <cr>	read <message> timeout	illegal pager for this operator
write sysoutput status	read <message> status	checksum error
invalid logon string	read <message> invalid length	illegal dest_city/local or broadcast
write <RS> status	read <message> no <CR>	illegal dest_city/caller selects
invalid password	read <city> timeout	message too long – truncated
activity timeout	read <city> status	illegal characters in the message
logon timeout	read <city> invalid length	write <ACK><CR> status
inter char timeout	read <city> no <CR>	write <RS><CR> status
read <STX> status	read <ETX><checksum> timeout	logon discards
non <STX>/<EOT> received	read <ETX><checksum> status	<EOT>'S at logon
time out on <CR> after <EOT>	read <ETX><checksum> length	date changes
read <CR> after <STX> status	Enhanced Service Function Setup Incomplete	
	Maximum Enhanced Services Currently Active	

A skymds02 log-file example is:

```
-----  
STATS SINCE:- 05/08/2002 02:10:03  
75 log on  
75 log off  
75 message received  
1 date changes  
06/08/2002 02:10:02 date changed stats reset  
-----
```

```
STATS SINCE:- 06/08/2002 02:10:02  
49 log on  
48 log off  
49 message received  
2 non <STX>/<EOT> received  
1 date changes  
07/08/2002 02:10:01 date changed stats reset  
-----
```

```
STATS SINCE:- 07/08/2002 02:10:01  
65 log on  
65 log off  
65 message received  
1 date changes  
08/08/2002 02:10:05 date changed stats reset  
-----
```

13.6.3 LOG MESSAGES WRITTEN TO MDSLOG.LOG (AND USUALLY STDERR)

Many <System_Name> programs use the mds_log() utility to permanently write log messages to a file. Presuming that the program calling mds_log() has set the variable ‘mds_data_dir’ to equal /var/mds/data – the usual location of <System_Name> data files and the usual value of the MDS_DATA_DIR environment variable – the messages written are duplicated to both /var/mds/data/mdslog.log and stderr. Environment variables may disable one or both of these message locations.

Couldn't understand the PET logon

SYSTEM MALFUNCTION

Couldn't write the ID string to the communication port

**ERROR=06 write ID
 failed, errno=<num>, tmsfr=<num>**

Invalid password

Invalid passw attempt - pwd=< 6-character string >

The variable MDS_PET_PG?_ACTIV_TOUT was not found by skymds02 when looking in its environment. This value is set by the start.mds script that executes skymds02, so this messages should not ever be seen

ERROR=01 no env variable for MDS_PET_PG?_ACTIV_TOUT

Couldn't send a recognition string back to the connecting computer over the communications port

ERROR=07 write lun_comms <recognition-string> failed, errno=<errnum>, tmsfr=<how much written >

Testing security for sending SMS messages

Invalid SMS security group=<SMS? >

... and ...

Pa=< pager-ID > not allow to page external.

Couldn't find the port record in the ports table

isread(IEQUAL) < MDSPORTS > failed iserrno=< MDSPORTS >

isrewcurr() < MDSPORTS > failed iserrno=< MDSPORTS >

couldn't set the operator name in the ports record in the database

Some classes of accounts can't send to external ‘contacts’

Security <SMS?> not allow to page external contacts pa=< pager number >

Checksum seems to be wrong

***** 36 checksum error, logon=< current_operator >**

Couldn't close the communication port

close() failed, errno=< errno > ;

Couldn't open the communication port

Open of < comms_tty >.16s failed, errno=< comms_tty >

This program may already be running) ;

Problems opening / creating the shared memory where the last-activity time-stamp is kept (to detect inactivity)

could not create shared memory

cannot shmat memory

Could not open the bulk queue

Line < _LINE_ >: queue_open() failed

Couldn't access and lock the port to communicate with the xacom telecommunications server

Lock of xacom comms port failed

This program may already be running

The baud rate set in the ports-record (in the C-ISAM database) does not seem reasonable

invalid baud rate

In stat_opers, could not open either a terminal or log-file to dump our log messages to
Could not open < datname > errno=< datname >

In validate_user, a problem with the security code
Invalid security code=< sec-code > for usercode=<user-code >

Couldn't (1) read the password, (2) find the port record, (3) write the port record
isread() < MDSPASSW > failed iserrno=< MDSPASSW >
isread(ISEQUAL) < MDSPORTS > failed iserrno=< MDSPORTS >
isnewcurr() < MDSPORTS > failed iserrno=< MDSPORTS >

Couldn't get the two-letter destination queue name from the "dests" database table
Open of < MDSDESTIM>; > failed
isstart() of < MDSDESTIM > failed

13.7 TERMINATING SKYMD02

13.7.1 SEND THE SIG_TERM SIGNAL

This is actually caught, but the variable "termination_requested" is set to true. This allows skymds02 to exit in a dignified manner. termination_requested is tested:

- during the validation of each new user
- at the start of waiting for each message
- every time the modem is hung-up (usually in a hang-up – reconnect loop)

Unlike many <System_Name> programs, skymds02 does not listen to the termination semaphore

13.7.2 KILL -9 PID

Kill the program externally.

13.7.3 LOG-ON WITH AN "EXIT" PAGER NUMBER

This will be read in get_valid_message() and the program terminated.

Invalid parameters passed to skymds02 on its command-line will terminate the new instantiation.

13.7.4 OTHER

The program will terminate if enough junk is sent to its input stream

14 skymds70

14.1 SKYMDS70 OVERVIEW

skymds70 implements the MERP interface for the connection of (remote) customer computers to the <System_Name> system. MERP (Message Entry and Retrieval Protocol) receives messages from remote computers and adds them into Bulk Queue for transmission to their destination, which may be a mobile phone, a pager, an email box, a fax machine, or (possibly) SAGR²⁷. An alternate protocol is pager entry protocol (PET), implemented by skymds02 (*q.v.*)

Customers using <System_Name> over one of three forms of hardware connection may utilise the MERP protocol. Larger customers use leased digital line: Telstra's "DMS" – Digital Metropolitan Service. Smaller customers may either use a normal telephone line with a normal modem, or, as a cheap alternative to DMS, use "camp-on" modems, normal modems attached to leased lines.

14.2 RUNNING SKYMDS70

skymds70 is started from the start-up UNIX script /opt/mds/bin/start.mds, multiple copies running on each server.

14.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-t <comms tty>	The port for the program to run on
-c	Do not check the SMS security
-D	Run in debug mode, listing detailed run-time information.
-s <seconds>	Sleep time between processing each message. default = 0
-x	Run in 'ixon' mode (see following description).
-o	Run as MERP version 1
-n	Run in numeric destination mode
?	Print the usage instructions

Table 92: skymds70: Command-Line Parameters

14.2.2 EXAMPLE COMMAND LINE

The command line used to execute skymds70 at one <System_Name> site is:

```
$MDS_BIN_DIR/skymds70 -x -s 2 -t /dev/syd18 2>$LOG_SCR
```

indicating that skymds70 is being run:

- in ixon mode (see following description).
- with sleep time of 2 seconds,
- on the port /dev/syd18,

and that standard error (file descriptor 2) is being redirected to the stream that has been defined as LOG_SCR, presumably a terminal screen somewhere. MDS_BIN_DIR is usually set to /opt/mds/bin in the environment settings file start.env.

14.2.3 NOTE RE: IXON AND IXOFF.

The '-x' flag, passed on the command-line, indicates that both the IXON and IXOFF flags should be set. IXON signifies 'output control', any STOP signal received suspends output (back to the MERP connection customer). IXOFF is 'input control'. The HP-UX operating system will itself send a STOP signal down the serial port when its internal input buffer is about to overflow. This STOP signal will, hopefully, indicate to the MERP customer's computer system to stop or slow its message transmission.

Both IXON and IXOFF are set as a result of receiving the '-x' flag. While this is referred to internally as "ixon", it is the IXOFF flag (input control) that will be of most use to a heavily laden skymds70 daemon.

²⁷ SAGR: the South Australian Government Radio Network

14.3 PROGRAM DESCRIPTION

As detailed above, skymds70 implements MERP protocol connections by external computers to the <System_Name> system.

The flow of activity is controlled by skymds70's `main()` function. After setting up its environment and files (with `initialise()`) and calling `validate_user()` to check that an authorised user has run this daemon in the first place, `main()` enters an infinite loop, reading inputs – messages usually – from the connected computer and optionally writing the message to the bulk queue. More detail appears in the description of the `main()` function below.

14.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds70.

14.4.1 SOURCE FILES

Source file	Description
skymds70.c	Receive and send messages from/to remote computers using the MERP protocol
mdsqmain.c	Generic message queuing routine
gensubs.c	Contains various general purpose routines that are separately compiled
isstart.c	Convert the key definition string into the c-ISAM structure
mdssubs.c	Contains various general purpose routines
mdslog.c	Generic error logging
mdssemck.c	Check the nominated semaphore and return TRUE if it has dropped
mdscomio.c	Various serial IO routines
crccalc.c	Calculate the CRC for a given string
maskpagr.c	Read pager file with line file driven masking
security.c	Various Security Routines
mdsginit.c	A number of functions which perform commonly used initialisation code segments
mdsportn.c	Routines associated with the ports file
mdsqmnt.c	Queues file updates for x mission to other machines
mdssetup.c	This module contains routines to obtain its set-up definitions from a text set-up file
mdskbld.c	Contains functions to build and un-build pager keys
mdscdr.c	Contains various general purpose routines
mdstmrtn.c	Contains functions which perform operations to do with time
mdsqurtm.c	Generic message queuing routine
mdsemqmain.c	Generic message email queuing routine

Table 93: skymds70: Source Files

14.4.2 HEADER FILES (NON-DATABASE)

Header File	Description
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.: C time field definitions
macros.h	User defined macros.: definitions & macros for common functions
mdsaprec.h	A-Party Master file
mdscdr.h	CDR definitions file
mdstype.h	Used for validating message chars.
mdsflsz.h	Standard <System_Name> includes.: field len & definitions
security.h	Structure and defines for security groups

Table 94: skymds70: Header Files

14.4.3 DATABASE TABLES

Database table	Header File	Description
mdsacrec	mdscdrec.h	Alternative call details master
mdsaparty	mdsaprec.h	mds A-Party customer details
mdsbull	mdsbtrec.h	Operator bulletin
mdscdrec	mdscdrec.h	Call details master
mdsdestm	mdsderec.h	Destinations master
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdspassw	mdspsec.h	Password username
mdsports	mdsporec.h	Ports security
mdsqqb.txt	mdsqarec.h	Bulk queue
mdsqxx.txt	mdsqrec.h	Destination queue s
mdssecur	mdsserec.h	Security codes

Table 95: skymds70: Database Tables

14.4.4 ENVIRONMENTAL VARIABLES REFERENCED

Environmental Variable	Description
MDS_DATA_DIR	The root of the data directories, queues, database and history
MDS_MACHINE_ID	M,O for Melbourne, S,X for Sydney, A for Adelaide, B for Brisbane, P for Perth, and C for Canberra

Table 96: skymds70: Environment Variables

14.5 SKYMD70: MAJOR FUNCTIONS

14.5.1 FUNCTION: MAIN()

In skymds70, the main function contains the central logic of the program. It passes through the following steps:

- Do initialisation by using the `initialise()` function

Initialise completes tasks such as reading the relevant environmental variables, parse the command line input and opening all needed database tables.

- Validate the MERP user by calling `validate_user()`
- Initialise statistics logging by an initial call to `stat_oper()`
- Start the infinite loop

Inside this loop, as described, MERP format messages are collected from the external connection and passed to the <System_Name> bulk queue.

- should I shut down? Test the `shut_down` semaphore.
- call `get_valid_message()` to read the next message, polling the communications port until a message is received.
- Process the retrieved message

The message, stored in a global buffer, is processed by one of five stretches of code, dependent on its message type. The five processing options are the following:

1. Pager Message:
Call `validate_page_message()` to check the message before using `queue_main()` to send it to the bulk queue.

2. “Are you alive”
Sends either a pre-formatted message (the bulletin) or the date and time back to a connected computer that is testing that this skymds70 program is still operational.
 3. Cap-Code message (same as pager message)
One can send messages to a pager’s “Cap-Code” (unique ID), instead of its pager number. The function `validate_cap_message()` – called to handle messages of this type – is virtually identical to the `validate_pager_message()` functional called to handle pager messages. After the message has been checked, `queue_main()` is called to post it to the bulk queue.
 4. Message Enquiry
This option is not supported. `do_enquiry()` just returns, or prints out an error code in debug mode.
 5. Reset the Sequence number
The sequence number is reset to 1
- Return to the next retrieve message → process it loop

14.5.2 FUNCTION: INITIALISE

skymds70’s `initialise()` function is similar to the `initialise()` function in most other <System_Name> programs, but with the addition of code to establish serial port parameters.

- read in environment variables
- parse the command-line
- put skymds70 into the background
- set signal handlers for a few signals

Ctrl-C is caught, and handlers are set for SIGALRM, SIGUSR2 (debug mode signal), and termination

- Establish a serial connection to the port passed to skymds70 on its command line
- Adjust the serial port parameters as appropriate

The serial port settings are read into a data structure, the values of this structure adjusted as appropriate, and it is written back to the port²⁸ as its new settings.

- Attach to a shared memory block in order to record activity and inactivity.

The daemon mdsalarm monitors many parts of a <System_Name> installation. It monitors the inactivity of external connections by reading a shared memory block into which a last-write timestamp is written. When this time stamp is, say, 20 minutes old, an alarm is raised. skymds70 connects to the shared memory block in order to write to the shared memory.

- All the database tables needed are opened and an attachment is made to the bulk queue.
- The line table (mdsline) is checked for a valid line (indial setup) for the port passed on the command-line.

²⁸ using the system Function: `ioctl(post, TCSETA, data-structure);`

14.5.3 FUNCTION: GET_VALID_MESSAGE

get_valid_message() performs the dual tasks of both waiting for a connection to be made and testing and validating a message from that connection, once it is made. It runs in an infinite loop, reading characters from the serial port until both a valid connection sequence is detected and a valid message is delivered. It then breaks from the loop and returns to its caller, the **main()** function, delivering the message for future processing.

- read the serial port input into a buffer

The port is read until a CR is found, which indicates the end of a message. If no CR is found (i.e. the read times out), this bit of code loops, until a CR is, in fact, read. The read bytes are then scanned from the end backwards for a STX character, indicating the start of a message, meaning that if multiple messages are sent, not separated with a STX, then the previous message will be discarded. If STX isn't found the data is dumped and **get_valid_message()** goes back to reading for a CR character again.

- the CRC (Cyclic Redundancy Check – a check number) is calculated for comparison to the one in the message
- If the message is in the old MERP format, it is converted to the new MERP format
- A small amount of processing is then done on the retrieved message.

If the message is a Pager (or a Cap message – a pager message addressed straight to the Cap-ID of someone's pager), the CRC value embedded in the message is copied into a variable and the trailing blanks are stripped from the message. If the message is an “Are you Alive” test, a message enquiry²⁹, or an instruction to reset the sequence number, only the CRC value is copied. Next the message is tested for an ETX (end transmission) character and a correct CRC value compared to the value just calculated.

- If this is a duplicate of the previous message, with the same sequence number, just send the previous reply
- The message is saved and **get_valid_message()** exits

14.5.4 FUNCTION: VALIDATE_PAGER_MESSAGE

validate_pager_message() checks the destination pager number is valid and the message is OK. It is nearly identical to **validate_cap_message()**; cap and pager messages are the same things, just addressed differently.

- the status of the pager account is checked, using **check_pager_and_security()**
- compare the message length to the maximum length allowed for this pager type and the limit in the database
- convert any illegal characters to spaces.
- Prepare and check the destination city for the message. If an error, user **send_rs()** to return the error digit ‘2’ to the connecting process, and return FALSE, effectively discarding this message
- If got to here the message must be OK. Save it.
- If a sleep_time value has been passed on the command-line, then sleep for that number of seconds.
- send a error character ‘4’ back to the calling process if the message was too long, else send a ‘3’ to indicate that everything was OK

14.5.5 FUNCTION: VALIDATE_CAP_MESSAGE

Virtually identical to the **validate_pager_message()** above.

14.5.6 FUNCTION: QUEUE_MAIN

queue_main(), coded in mdsqmain.c and compiled into a separate module (mdsqmain.o), is called to add the message to the Bulk Queue.

²⁹ Pager Enquiries are not currently supported

14.5.7 FUNCTION: ARE_YOU_ALIVE

`are_you_alive()` is used to send a message back the party connected to the MERP connection in order to indicate that this process is still operational.

- the mdsbull table record matching port-type ‘N’ and city ‘B’ is looked for, and the text bulletin in that record extracted
- if the appropriate record was found, that text become the bulletin, else a time and date stamp is used.
- The rest of the record is formatted, such that it makes a valid serial port MERP transaction message
- The record is written to the serial port.

14.5.8 FUNCTION: STAT_OPERS

`skymds70` groups its statistical reporting tasks in one the one Function: `stat_ops()`. These tasks are of two types.

1. Note that an event has happened: Initialisation, logon, logoff
2. Respond to an event, and write all collected statistic to a log file. The list of log messages written appears in the Log Messages section. ‘Enquiry’, ‘Date_change’, and ‘Finalise’ are examples of this.

For all actions except ‘Enquiry’, the log messages are written to a file named: `MDS_DAT_DIR/logs/mdspet???.txt`, e.g. `/var/mds/data/logs/mdspet24.txt`. Enquiries are written to `stderr`.

14.5.9 OTHER FUNCTIONS

14.5.9.1 validate_user

Using the password and security database tables (`mdssecur` and `mdspassw`), the customer running the MERP connection is checked for the security to post pager messages.

14.5.9.2 check_pager_and_security

Checks that the pager exists and is properly set up, or that the customer can send SMS messages if that is what they are doing. Also test whether, if sending to a pager, it is one of the pagers permissible for this sender: e.g. belonging to the same company.

14.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to `stderr`
- Those only written to `stderr` while in debug mode.

The log messages produced by `skymds96a` are listed following:

14.7 STOPPING SKYMDS70

1. Set the `shutdown_semid` to no-zero
2. Find its process id and kill –9 it (as root)

15 skymds18

15.1 SKYMD18 OVERVIEW

skymds18 send messages into <System_Name> from <Company_Name>'s billing system, the Wang-based 'SKY<COMPANY_NAME>'. Basically it casts streams of data arriving on the communication port into a data structure, reads a 'what-am-I' field in this data structure, before passing the collected data to the appropriate function (from 30 or so) for processing *in* or manipulation *of* the <System_Name> system. Its partner program, skymds25, sends message from <System_Name> to the Wang.

SKY<COMPANY_NAME> is in Sydney; thus skymds18 only runs on lnks. Skymds25 is also only found in Sydney.

15.2 RUNNING SKYMD18

skymds18 is run from the start.mds script on the Sydney lnks <System_Name> server (only).

15.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-t <comms tty>	The communications port on which to read messages from SKY<COMPANY_NAME> Typically -t /dev/x25016
-D	run in debug mode, listing detailed run-time information.
-?	Print simple command-line usage instructions

Table 97: skymds18: Command-Line Parameters

15.2.2 EXAMPLE COMMAND-LINE

First the user / system administrator running skymds18 is asked:

```
Start skymds11, skymds67, skymds18, skymds25, skymds28, skymds81,"  
skymds06, skymds27, skymds10, remterm and skymds20 (y/n) : "
```

If they answer 'y', the listed programs are run, including skymds18, which is executed by the line:

```
$MDS_BIN_DIR/skymds18 -t /dev/x25016 2>$LOG_SCR
```

indicating that skymds18, residing in the directory MDS_BIN_DIR (/opt/mds/bin/) is going to read from port /dev/x25016 and have its stderr stream (file descriptor 2) duplicated to LOG_SCR.(defined in start.env to be /dev/tty1p0.)

15.3 PROGRAM DESCRIPTION

As noted above, skymds18 reads messages from SKY<COMPANY_NAME> via a communications port, casts the data stream read into a structure and process this structure in one of 30 or so ways depending on the type identifier embedded in the data-stream. Each complete transmission is collected by **get_valid_message()** and then passed to the appropriate function from 30 possibilities.

15.4 PROGRAM STRUCTURE

Following are listings of the components that have gone into skymds18.

15.4.1 SOURCES FILES

Source File	Description
skymds18.c	Interface to SKY<COMPANY_NAME> (Wang) stock and billing computer to receive test messages and file maintenance.
crcalc.c	Calculate the CRC for a given string.
gensubs.c	Contains various general-purpose routines that are separately compiled.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
maskpagr.c	Read pager file with line file driven masking.
mdsaurtn.c	Audit trail.
mdscdr.c	Contains various general-purpose routines that are separately compiled.
mdsemqmain.c	Generic message email queuing routine.
mdsginit.c	A number of functions, which perform commonly, used initialisation code segments.
mdskbld.c	Contains functions to build and unbuild pager keys, and set-up a dummy pager.
mdslog.c	Generic error logging.
mdslread.c	Line file read functions during Austel Number Change.
mdsnormn.c	Generate normalised name from holder_name.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurn.c	Generic message queuing routine.
mdsseck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssetup.c	Obtain set-up definitions from a text set-up file.
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions, which perform operations to do with time.
mdsvalpg.c	Validate pager versus operator.
security.c	Various Security Routines.

Table 98: skymds18: Source Files

15.4.2 HEADER FILES

Header File	Description
ascii.h	Function keys and special chars.: PMS function key values.
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.: C time field definitions
macros.h	User defined macros.: definitions & macros for common functions
mdsaprec.h	A-Party Master file
mdsaurec.h	Audit trail master file
mdscdr.h	CDR definitions file
mdscdrec.h	CDR data structure
mdsclrec.h	Client master file
mdscorec.h	(CLT/INV/GRP) Contacts Master File
mdscstype.h	Used for validating message chars.
mdsderec.h	Ports security master file
mdsecrec.h	External Contact master file
mdsemrec.h	Email File (Online & Batch)
mdsfarec.h	Pager facilities master file
mdsfldsz.h	Standard <System_Name> includes.: field length & definitions
mdsfxrec.h	(FAX) Fax Master File
mdslirec.h	Needed for the acdmem.h LINEMBX structure: History primary file
mdsparam.h	parameters used by PMS system
mdsparec.h	Pager Master File
mdsporec.h	Ports security master file: Ports Record
mdpsrec.h	Password master file
mdsqarec.h	Encoder queue: Message queue file
mdsqrec.h	Contacts Message queue file: Destination Message queue file
mdsserec.h	Security Group master file
mdstmrec.h	Temporary message master file
mdswarec.h	Wang interface comms layout
security.h	Structure and defines for security groups

Table 99: skymds18: Header Files

15.4.3 DATABASE TABLES

Each of these header files contain (only) a C-structure that matched the structure of a database table used by skymds18.

Database table	Header File	Description
mdsaparty		A-Party Customer Info
mdsaudit	mdsaurec.h	Audit trail
mdscdrec	mdscdrec.h	Call details master
mdsclien	mdsclrec.h	Client master
mdsconta	mdscorec.h	Contacts master
mdsdestm	mdsderec.h	Destinations master
mdsextco	mdsecrc.h	External contact master
mdsemail	mdsemrec.h	e-mail master
mdsfacil	mdsfarec.h	Facilities – what special <System_Name> services can be used
mdsfax	mdsfxrec.h	Fax master
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdspassw	mdpsrec.h	Password username
mdsqqb.txt	mdsqarec.h	Bulk queue
mdsqxx.txt	mdsquarec.h	Destination queues
mdssecur	mdsserec.h	Security codes
mdstmesg	mdstmrec.h	Temporary message master

Table 100: skymds18: Database Tables

15.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description
MDS_DATA_DIR	Where the data directory is, usually /var/mds/data
MDS_MACHINE_ID	What machine are we on: B=Brisbane, M,O = Melbourne, O = Perth ...

Table 101: skymds18: Database Tables

15.5 SKYMD18: MAJOR FUNCTIONS

15.5.1 FUNCTION: MAIN

The main input / output loop is implemented in the main() function itself. Here the data is read from the Wang (SKY<COMPANY_NAME>) by get_valid_message(), and passed off to various parts of the <System_Name> system, both as maintenance updates and as <System_Name> messages. So ...

- first initialise() is used to set up the program environment
- stat_oper(), with the parameter INITIALISE, is used to prepare the statistics logging system
- Now the infinite loop starts. Termination is tested at the end of each loop.
- get_valid_message() is used to get, as the name suggests, the next message from the communications port. The shutdown_semaphore is also tested in get_valid_message().
- A copy of the message with a timestamp are written to the **MDS_DATA_DIR/logs/mdswang.aud** file

The records written look something like:

20020917021924	15052173212L1D0000069626492705001556720	000000000035002043
ANOPAL	244307SYD	JUSTIN KELLY
SYDNW		0160 I000N 002
		0

Timestamp

These daily audit files grow to 10-15 MB; there is quite a lot of traffic between the Wang and <System_Name>.

- Now, in a long switch statement, the data collected by the `get_valid_message()` function is passed to one of 30 odd handler functions, the switch selection being made by the return value ('`tran_type`') from `get_valid_message()`. The `tran_type` obviously represents what type of message or update is arriving from the Wang.

The functions called are as follows. See below for a fuller description.

<code>tran_type</code>	Function Called
<code>TEST_PAGE_NUM</code>	<code>proc_test_page_num(mess_len);</code>
<code>TEST_PAGE_CAP</code>	<code>proc_test_page_cap(mess_len);</code>
<code>PAGER_ADD</code>	<code>proc_pager_add();</code>
<code>PAGER_CHANGE</code>	<code>proc_pager_change();</code>
<code>PAGER_DELETE</code>	<code>proc_pager_delete();</code> No response (no handler).
<code>TEMP_DUMMY_TRAN</code>	
<code>LINE_ADD</code>	<code>proc_line_add();</code>
<code>LINE_CHANGE</code>	<code>proc_line_change();</code>
<code>LINE_DELETE</code>	<code>proc_line_delete();</code>
<code>LINE_PAGER_ADD</code>	No response (no handler).
<code>LINE_PAGER_CHANGE</code>	No response (no handler).
<code>LINE_PAGER_DELETE</code>	No response (no handler).
<code>PAGER_GROUP_ADD</code>	<code>proc_pager_group_add();</code>
<code>PAGER_GROUP_CHANGE</code>	<code>proc_pager_group_change();</code>
<code>PAGER_GROUP_DELETE</code>	<code>proc_pager_group_delete();</code>
<code>CLIENT_ADD</code>	<code>proc_client_add();</code>
<code>CLIENT_CHANGE</code>	<code>proc_client_change();</code>
<code>CLIENT_DELETE</code>	<code>proc_client_delete();</code>
<code>PIN_ADD</code>	<code>proc_pin_add();</code>
<code>PIN_CHANGE</code>	<code>proc_pin_change();</code>
<code>PIN_DELETE</code>	<code>proc_pin_delete();</code>
<code>EMAIL_ADD</code>	<code>proc_email_add();</code>
<code>EMAIL_CHANGE</code>	<code>proc_email_change();</code>
<code>EMAIL_DELETE</code>	<code>proc_email_delete();</code>
<code>DELAY_PAGER_ADD</code>	<code>proc_delay_pager_add();</code>
<code>DELAY_PAGER_CHANGE</code>	<code>proc_delay_pager_change();</code>
<code>DELAY_PAGER_DELETE</code>	<code>proc_delay_pager_delete();</code>
<code>APARTY_ADD</code>	<code>proc_aparty_add();</code>
<code>APARTY_CHANGE</code>	<code>proc_aparty_change();</code>
<code>APARTY_DELETE</code>	<code>proc_aparty_delete();</code>

Table 102: skymds18: Functions used to send messages to SKY<COMPANY_NAME>

- and acknowledge message is sent back to the Wang along with a sequence number (between 1 and 9)
- finally the shutdown semaphore is tested to see whether to exit the skymds18 program.

15.5.2 FUNCTION: INITIALISE

The initialise() function sets up skymds18's run-time environment, via the following steps:

- Read environment variables (`MDS_DATA_DIR` and `MDS_MACHINE_ID`) from the environment
- Parse the command line
- call background to put the process in the background
- catch a few signals
- call `PORT_set_port_started()` to flag in the port record (in the mdsports database table) that the port is being used, the program has started.
- open the communication port (as a file)
- turn the non-blocking status of the communications port off and lock the port for our exclusive use

“blocking” means “wait-ininitely for input”. Turning `non-blocking` off (a double negative) ensures that the file descriptor for the communications port will wait for input; it won't expect everything to be on time.

- the port parameters are set correctly

`ioctl()` is used to read the parameters into a structure, these parameters are set and adjusted, before `ioctl()` is used to write the parameters back to the port.

- the shutdown_semaphore is accessed
- All the database tables we need are now opened, as are the maintenance queue (for inter-city updates and synchronisation) and the log file (for records of each transaction)
- attach to the security shared memory segment
- call `validate_user()` to validate the person who started `skymds18`, and record that the program has been started.

15.5.3 FUNCTION: GET_VALID_MESSAGE

`get_valid_message()` retrieves and pre-processes data (messages) sent from the Wang via the communications port.

- first it checks the shut_down semaphore
- reads the port
- check that the first character in the read data is STX

STX means Start of Text (ASCII character 2). If the first character is not STX, dump the message and send a NAK (negative acknowledge) to the WANG.

- check that the message is not too long, that it terminates with a CR (carriage return , ASCII 13), that the 6th last character is an ETX (End of Text, ASCII character 3)
- call `crccalc()` to generate the CRC, and compare this CRC to the one embedded in the message (past 5 characters)
- check that the sequence number on the message is the same as the expected sequence number.

If it is not, then reset the sequence number to 0, and call `send_ack()` to transmit this new sequence number to the Wang. If the sequence number is OK, pass the correct `sequence_number` and the previous result back the `SKY<COMPANY_NAME>` instead.

- Now read the `tran_type` field (bits 13 to 15) and set an internal `skymds18` constant based on this

e.g. the `tran_type` Code ‘EC’ sets the internal `tran_type` constant to be `EMAIL_CHANGE`. This is the value returned from `get_valid_message()` and determines how the remainder of `skymds18` processes the message.

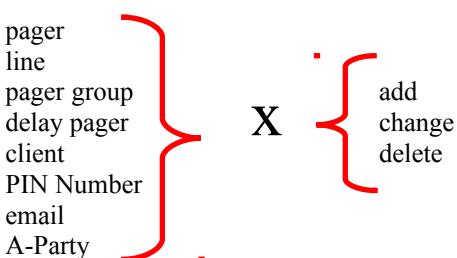
- finally the last 6 characters and up to character 255-6 (i.e. 249) are set to spaces.

The structure (union) holding the message is 300 characters long, while the longest current use is 255 characters. This deletes anything between the end of the message currently stored in the buffer (which may be much shorter than 255 bytes) and the ‘255-6’ byte mark.

15.5.4 FUNCTIONS: THE 28 WANG MESSAGE PROCESSING FUNCTIONS (IN BRIEF)

Here are short descriptions of each of the 28 `SKY<COMPANY_NAME>`-message-processing function.

Diagrammatically:



plus testing a pager number and ‘cap code’, the unique pager identifiers.

15.5.4.1 `proc_test_page_num(mess_len)`

This function posts a message to the <System_Name> bulk queue.

- After extracting the pager number from the message, `proc_test_page_num()` passes it to `validate_pager()` to ascertain the status of the pager message.
- If validate `pager()` indicates that the pager message is not found, out of service, is <Company_Name>ed to an invalid client or has invalid security `proc_test_page_num()` returns with an error indicating this.
- Next the message text itself is validated
- If the message (and everything else) is OK, `setup_queue_rec()` and `queue_main()` are called to prepare the message and pass it to the <System_Name> bulk queue.

15.5.4.2 `proc_test_page_cap(mess_len)`

`proc_test_page_cap()` is very similar to the previous `proc_test_page_num()`.

15.5.4.3 `proc_pager_add()`

This function is used to add a pager to the <System_Name> system as a result of a Wang message.

- the pager record is set up with specialised information from the Wang message, following which `setup_pager_rec()` is used to construct the remaining parts of the record
- add the appropriate security to the pager record
- try to add the pager record to the mdspager database table. If there is already a pager with the same pager number it is written over, otherwise the new pager record is added as normal
- after either adding or updating the record a message is also written to the maintenance queue to add or update all the other <System_Name> systems
- now the field that indicates the number of pagers with the same cap_code³⁰ is updated and, similarly, the number of other pagers with the same client code is updated. Both of these changes are also duplicated via the maintenance queue.

15.5.4.4 `proc_pager_change()`

`proc_pager_change()` performs a very similar task to `proc_pager_add()`, and, in fact, `proc_pager_add()` can (it seems) accomplish pager updates itself – if an add is sent through with the same pager number.

- first mdspager table is searched for the pager to update. If no record is found `proc_pager_add()` is used to add (c.f. update) the pager to the mdspager table.
- set-up `pager_rec()` is now used to prepare the (updated) pager record
- as we are already on the pager record to update – it was found in the first step above – so now just write the updated pager record over the top
- call `write_audit()` to write an audit record.
- the number of pager records with the same cap_code (unique physical pager identifier) as this updated record is written to this updated record. There may be three or more mdspager records concerning the same physical pager.
- if the client code sent from SKY<COMPANY_NAME> / the Wang doesn't match one already existing in the client database table (mdsclien), a dummy client number is added to the record.

³⁰ The cap_code is the unique identifier for each physical pager. Multiple pager records with the same pager number indicate that messages to what seem like different pagers, well, at least different pager numbers are actually going to the same (physical) pager.

15.5.4.5 proc_pager_delete()

As the name suggests **proc_pager_delete()** deletes pager records from the pager table. It actually just changes the pager_status to INACTIVE_STATUS, but you get the idea.

- first it finds the record it is about to delete. If the record doesn't exist then exit.
- also exit if the pager SKY<COMPANY_NAME> is trying to delete is an SMS cellphone, as the Wang is not allowed to delete these
- change the status to INACTIVE_STATUS, effectively deleting the record
- write the new record to the maintenance queue to distribute the messages to the other <System_Name> servers around Australia
- decrement the number of pagers with the same client on all the other pager records that have the same client (as this just-deleted) record.
- and now the record is posted to all the other <System_Name> servers again!

15.5.4.6 proc_line_add()

Provisioning: add a new indial (line) record to the <System_Name> system.

- **setup_line_rec()** prepares a new mdsline record, read all the information from the data skymds18 has just received from SKY<COMPANY_NAME>. Once it returns, **proc_line_add()** clears a few extra fields.
- an attempt is made to write the new line record to the mdsline database table
- if there is already a record with the same indial this new record is written over the top of it. Again it displays the change functionality that there is a separate function (**proc_line_change()**) to perform.
- the addition is transmitted to the other <System_Name> servers via the maintenance queue

15.5.4.7 proc_line_change()

Update an existing line record.

- first find if there actually is a record with the line number of the record that we are trying to update. If not, then use **proc_line_add()** to add the record.
- **setup_line_rec()** prepares the new line record from the SKY<COMPANY_NAME> data
- this new record is now written over the top of the existing record with the same line number
- If the client specified by the Wang data does not exist (check the mdsclien table) then write a dummy client to the updated record.
- Check the contacts (advanced processing services) set up for this indial. If the service type on the updated record is either BMP (?) or PMP (?), and if the contact record for this pager has a type of PAG and a pager number of 000000, then delete the contact from the contact table and transmit this update to the other <System_Name> servers via the maintenance queue(s). Not sure exactly why

15.5.4.8 proc_line_delete()

Delete a line record.

- find the pager record with the line number as stipulated by the message from SKY<COMPANY_NAME>
- if the service attached to this line number happens to be SMS, then exit from **proc_line_delete()** as SMS services are not to be deleted from the Wang
- mark the line as inactive, effectively deleting it
- write to the maintenance queue this update

15.5.4.9 proc_pager_group_add()

Add a pager group to <System_Name> as specified by the message from SKY<COMPANY_NAME>.

- using find_pager_group_rec(), try to find a pager group attached to this pager number. If there does seem to be a group set up for this pager already, then transfer to proc_pager_group_change() to add to the existing group
- call setup_pager_group_rec() to, as the name suggests, set up a group pager record (using the data in sent from SKY<COMPANY_NAME>)
- write the new group to the facilities table mdsfacil and transmit the facilities addition to the other <System_Name> servers via the maintenance queue
- increment the number of pager numbers attached to this top-level pager number
- send change message to the maintenance queue, changing the data for this pager

15.5.4.10 proc_pager_group_change()

Change the membership of a pre-existing pager group.

- using find_pager_group_rec(), try to find a pager group attached to this pager number. If there does seem to be a group set up for this pager already, then transfer to proc_pager_group_change() to add to the existing group
- call setup_pager_group_rec() to set up a group pager record (using the data in sent from SKY<COMPANY_NAME>)
- write the new group record over the top of the existing facilities record
- increment the number of pager numbers attached to this top-level pager number
- send an update message to the maintenance queue, updating the data for this facility.

15.5.4.11 proc_pager_group_delete()

Delete a pager group.

- find_pager_group_rec() to locate the pager group to be deleted
- delete the pager group from the facilities database table and transmit the changes to the other <System_Name> servers via the maintenance queue.
- change the data for the pager number that represents the group, and transmit this change to the other <System_Name> servers via the maintenance queue

15.5.4.12 proc_client_add()

Add a client to the <System_Name> system

- call setup_client_rec() to prepare a client record with the data just received from SKY<COMPANY_NAME>
- write the client record to the mdsclient client database table. If a record with the same client number already exists in the table, replace that record with the new one
- write the update to the maintenance queue for replication to the other <System_Name> sites

15.5.4.13 proc_client_change()

Change a client record in the <System_Name> system

- find the client record (in mdsclient) with the same client number. If there isn't one, call proc_client_add() to add the client and exit.
- setup_client_rec() prepares the client record with the new data from the Wang
- write the record to the client table
- call mnt_queue_write() to copy the changes to the other <System_Name> servers

15.5.4.14 proc_client_delete()

Delete a client record from <System_Name>

- find the record to be deleted in the mdsclient record. If it's not there, well, obviously its time to exit
- delete the record
- call mnt_queue_write() to copy the changes to the other <System_Name> servers

15.5.4.15 proc_pin_add()

Add a PIN number to a <System_Name> pager record.

- get the pager record from the mdspager database table
- if the type of service attached to this pager is SMS³¹, then don't make any change. SKY<COMPANY_NAME> is not allowed to change SMS service records
- Add the PIN number to the pager, and replicate the change to the other <System_Name> servers via the maintenance queue

15.5.4.16 proc_pin_change()

Change a PIN number on a <System_Name> pager record.

- get the pager record from the mdspager database table. If its not there we obviously have a problem
- if the type of service attached to this pager is SMS³², then don't make any change. SKY<COMPANY_NAME> is not allowed to change SMS service records
- Change the PIN number for the pager, and replicate the change to the other <System_Name> servers via the maintenance queue

15.5.4.17 proc_pin_delete()

Ummm, delete a PIN number from a <System_Name> pager

- get the pager record from the mdspager database table. If its not there we obviously have a problem
- if the type of service attached to this pager is SMS³³, then don't make any change. SKY<COMPANY_NAME> is not allowed to change SMS service records
- Delete the PIN number from the pager, and replicate the change to the other <System_Name> servers via the maintenance queue

15.5.4.18 proc_email_add()

Add an email address to a <System_Name> pager record.

- get the pager record from the mdspager database table. If its not there we obviously have a problem
- call setup_email_rec() to prepare the email record.
- write the email data to the email database table (mdsemail) and replicate the change to the other <System_Name> servers via the maintenance queue

³¹ SMS cellphones can be "pagers" receiving SMS (pager) messages

³² SMS cellphones can be "pagers" receiving SMS (pager) messages

³³ SMS cellphones can be "pagers" receiving SMS (pager) messages

15.5.4.19 proc_email_change()

Change an email record for a pager

- get the pager record from the mdspager database table.
- get the email record for this pager
- if the email record doesn't exist, call proc_email_add() to add it
- call setup_email_rec() to prepare the email record.
- write the changed email data to the email database table (mdsemail) and replicate the change to the other <System_Name> servers via the maintenance queue

15.5.4.20 proc_email_delete()

Change an email record for a pager

- get the pager record from the mdspager database table.
- get the email record for this pager. If it doesn't exist then we're all done.
- delete the email record.

15.5.4.21 Note on the 'Delay Pager' Facility

From the facilities header file mdsfarec.h:

"Provides the ability for messages entered between certain times to not be delivered till a later time. If a message arrives inside the window of start_time and end_time the call will be 'booked' to be delivered at the time at the end of the window. e.g. from 08:30 to 19:30 any messages arriving in this period delivered at 19:30 "

15.5.4.22 proc_delay_pager_add()

Adds a 'delay pager' record to the facilities database table, mdsfacil.

- find the pager in the mdspager table
- checks that the pager is active
- see if there is a delay pager facility set up for this pager. If there is then call proc_delay_pager_change() instead. Note that there is nothing in <System_Name> to stop there being multiple delay pager records for a single pager; duplicates are allowed in the facilities table.
- setup_delay_pager_rec() to, well, set up the delay pager record.
- write the record to the facilities table.
- replicate the change in both the facilities and pager tables to the other <System_Name> servers with mnt_queue_write()

15.5.4.23 proc_delay_pager_change()

Change, update a delay pager record

- find the pager in the mdspager table
- checks that the pager is active
- see if there is a delay pager facility set up for this pager. If there isn't then call proc_delay_pager_add() instead.
- call setup_delay_pager_rec() to write the SKY<COMPANY_NAME> updated delay pager data to a facilities record.
- write the updated facilities record back to the mdsfacil facilities table
- replicate the change in both the facilities and pager tables to the other <System_Name> servers with mnt_queue_write()

15.5.4.24 proc_delay_pager_delete()

Delete a delay_pager record from the facilities table

- find the pager in the mdspager table
- checks that the pager is active
- find the appropriate record in the facilities table
- delete it, and use mnt_queue_write() to update the other <System_Name> servers
- Change the delay service flag on the pager record to N (No)
- transmit the updated pager record to the other pager records via the maintenance queue

15.5.4.25 proc_aparty_add()

Adds an A-Party customer to the <System_Name> system. A-Party customers pay for the messages they send (as opposed to the cost being covered by the receiver) and, as a result, are allowed access to advanced services such as sending Bulk email messages.

- call setup_aparty_rec() to set-up an A-Party customer record from the data transmitted from the Wang
- write it to the A-Party database table (mdsaparty)
- transmit to the change to the other <System_Name> servers via the maintenance queue

15.5.4.26 proc_aparty_change()

Update an A-Party customer's details.

- read the A-Party record from the database table, to both ensure that its there and to set it as the current record.
- call setup_pager_rec() to write a new A-Party record from the Wang data.
- write the new, updated record over the top of the previous record
- use mnt_queue_mnt() to transmit the update to the other <System_Name> servers

15.5.4.27 proc_aparty_delete()

And, finally, proc_aparty_delete() to delete an A-Party record.

- first setup_aparty_rec() is called to set up an A-Party record. This is only done to be able to use the record to find the equivalent record in the A-Party data base table (mdsaparty)
- delete the record
- transmit the changes to the other <System_Name> servers via the maintenance queue.

15.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

15.7 STOPPING SKYMD18

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)
3. If no messages are being sent from the Wang / SKY<COMPANY_NAME> then skynds18 is effectively non-operational. Not really stopped though.
4. Turn the <System_Name> server machine off at the wall (only joking, don't do this on any sort of regular basis)

<System_Name>: Processing and Maintenance

16 The history Server, History Data

16.1 OVERVIEW

The history server and the CDR record all message details. The history server records the details of every message, for both internal <System_Name> uses and to enable a message to be resent (if necessary). There is one History Server (lnkh) and it lives in Melbourne. Compared to history data CDR is billing focused, recording message length, subscriber type, priority, phone number and the like. As a traditional continuation of past practices, CDR data is gathered in Adelaide and passed back to the Oracle LMS (<Company_Name> Mediation System³⁴) in Melbourne for billing purposes.

Historical information is, to some extent, duplicated on each local server and the history server. The history data, both primary and secondary (see below), are held as per the table:

Date	local <System_Name> server	History Server by City (e.g. ...hist / M /)	History Server Combined (.../hist)
next 50 days	-	-	empty
next 7 days	empty	-	empty
dated today ³⁵	full	-	empty
to 7 days ago	full	-	empty
7 to 14 days ago	full	full	full
14 to 43 days ago	full	compressed	full
43 to 365 days ago	-	compressed	full

Table 103: History Data Availability

16.2 HISTORY

16.2.1 MACHINE – HISTORY SERVER

The History Server – lnkh ('h' for history), and its directories /var/mds/data/hist/? (A, B, M, P, and S for Adelaide, Brisbane, Melbourne, Perth and Sydney respectively) are the home of the <System_Name> history files. Local history files live on each of the other <System_Name> machines, in their own /var/mds/data/hist/ directory – not in parallel A,B,M,P,S directories.

16.2.2 HISTORY DATA

<System_Name> history exists in two supplementary forms: Primary and Secondary.

- Primary history, a daily C-ISAM database table (*.dat and *.idx pair), which holds all the information *about* the message, except the message id and the message text itself. That information is stored, separately, in the secondary history file. An example of a primary history file is (the pair) hi20020730.dat and hi20020730.idx
- Secondary History: a textfile (*.txt) accompanying the primary history database table, and containing the core message: its text and its message ID. Secondary history files are named as hs20020730.txt

As the examples show, history files are named:

Primary: hiccyymmdd.dat & hiccyymmdd.idx, e.g. hi20020730.dat & hi20020730.idx. NB: "hi" for primary
Secondary: hscyyymmdd.txt, e.g. hs20020730.txt. NB: "hs" for secondary

On each <System_Name> server, local history is stored in /var/mds/data/hist/. Here one will find the previous 40 days of primary and secondary history files, plus the next 7 days of empty history files waiting to be filled. Each <System_Name> system must have a week of empty history files to ensure that no records are ever lost.

³⁴ "mediation" is for billing

³⁵ Yes, today's history. Each History file is made during the small hours of the morning – containing the previous 24 hour's data. So the date on the most recent history file is, by the time you come to work, today's date.

On the history server, lnkh, is both the combined daily history files from each Australian site (in /var/mds/data/hist/), and compressed copies of each site's history files, a large collection, about 1 year's worth. The (compressed) primary history files are hiYYYYmmdd.dat.Z and hiYYYYmmdd.idx.Z, the secondary history file is the matching hsYYYYmmdd.dat.Z. All files older than 7 days are compressed, while those greater than a year are deleted.

16.3 HISTORY CONCEPTS

A few concepts that you, the reader, may not have encountered yet but that are necessary, really, for a proper understanding of the <System_Name> History system.

- As noted above, and displayed below, History is stored in two supplementary forms. The Primary history records, stored as an Informix database table, contain the central data about each message, while the secondary history holds the message text itself.
- Each message must have its own primary history record, as the primary records are used when recalling the messages from the History System.. Thus, if a 'Group' paging contact spawns five individual messages, there will be, in addition to a primary record for the group, five additional primary history records, one for each spawned message. If any of these messages themselves create further messages, further primary (and secondary) history records will be written for each. While the Group contact is the message originally sent, each additional message created also needs to be able to be identified by its primary history record.
- The top level primary history record has all the history records that it is ultimately responsible for creating recorded under it, that is, available and supplied when querying the history data. One <System_Name> message may create another (e.g. a pager message being diverted to a group) which may create another (one group "member" may be an escalation) which may create another (each message created by the escalation), etc. Each sub-level, that is all the messages except that first, original message, have a primary and appropriate secondary history record(s) only.
- Recent history is stored locally on each <System_Name> server. However the main history repository is a dedicate machine, the History Server 'lnkh', in Melbourne.

16.4 BRIEF REVIEW OF HISTORY FILE FORMATS

16.4.1 PRIMARY HISTORY RECORDS

Primary history records are the 'headers' for each set of secondary history records, containing the central information about the recorded transaction and the index / location of the first and last relevant history record.

C	TEST_GRP	232111	0096250296	D/dev/pts/tm	c	muoi	340001200	I 40	1726	2954
P	0000830497	232111	0096250296	D/dev/pts/tm	C	muoi	340001200	I 60	2000	2100
Contact / Pager	Contact name, or pager number	time	indial	port/device	machine	Transaction type	event number	source. I means 'Internal'	length of first	Secondary History - index of first

- index of last

16.4.2 CREATION AND HANDLING THE HISTORY FILES

Three scripts maintain the history files: start.68, start.hisrcp, and start.hismer. All of these run as overnight cron jobs. The three relevant crontab lines are the following:

```
5 2 * * *      /opt/mds/bin/start.68          #New History File Creator
0 3 * * *      /opt/mds/bin/start.hisrcp       #Copy history files
0 8 * * *      /opt/mds/bin/start.hismer        #Merge all history file
```

16.4.2.1 Creating History Files

The first cron Line instructs that at 2:05 am every morning, cron calls start.68, which, as the name suggests, starts the program skymds68. skymds68 creates new, empty history files, ensuring that on each server are always 7 days of empty history files available.

16.4.2.2 Storing History Files

Next cron (on lnkh) runs hiscrp to copy the previous day's history files from their respective locations on machines around Australia to itself .

16.4.2.3 Merging History Files

Finally, at 8am, start.hismer calls mdshismer to merge various collected history files together

16.4.3 FILLING, POPULATION OF THE HISTORY DATA FILES

During the day the empty history files are filled by:

skymds20 - The Bulk Queue to Delivery Queue process

The code to write skymds20 history details is in mdsqall.c. These functions write the secondary history record file (the message text and pager, SMS id, whatever) but only create (not write to) the more detailed primary history record.

Examples are:

- setup_and_write_hist_primary()** Set up the primary history file record and call hist_write_prim() to write it
- write_contact_parent_hsec_rec()** writes the file offset of the matching secondary history record (in the secondary history text file) onto the primary history record.
- hist_write_sec()** write the secondary history record

skymds03 - The distribution queue to delivery system daemon

For all messages except TMS, ADV (advisories), Batch Fax, Batch Email and Remote data a history record is written. "I got your message" type replies from a (remote) SMS centre also stimulate the writing of a history record. update_hist_file() and update_hist_by_remote_rec() are the functions in skymds03 used to write local and remote SMS history records respectively.

skymds67a – data back from SMSC

skymds67a processes the data received back from external SMS centres. However, the only information it writes to the history files is a note of which final carrier (e.g. Telstra, Vodafone, Optus) transported the SMS message. This also operates as an "I got your message" signal from the remote SMS centre.

16.4.4 PRIMARY HISTORY FILE (E.G. HI20020304.DAT)

The primary history file contains all the information about the call, and is described in complete detail in mdshirec.h.
As an example of a Primary History file record:

P00000005650030280038304628B/dev/vcz14 X JOYCEL q 0003 7000096666000096703

The call information held in this Primary History file record is as follows:

Field Value	Meaning
P	this is a pager call
565	the pager number
00:30.28	time message received
3830-4628	originating telephone number
B	code of the originating machine
/dev/vcz14	the port that the message was received on.
X	A repeat message. The codes are listed in mdshirec.h.
Joyce L	The operator that took the call
... a few fields not filled ...	
0003	the length of the transaction
7000096....	an index to matching records in the secondary history file.

Table 104: Meaning of Primary History file fields

16.4.5 THE SECONDARY HISTORY FILE (E.G. HI20020304.TXT)

The secondary history file contains the core data of the messages themselves, detailed in mdshsrec.h.

Examples of secondary history records are:

```
m 50010735827 WARWICK PLUMSTED, , , , PLS PH YOUR WIFE AT HOME
c 82010732034 SE20020708073011 DEPOT JAN M 512036300
d 0000null_ptr 07 s2 20020708073015 y1 20020708073013 a2 20020708073014 p2 2002070807 ... ...
r 60010732944200207080753320099631665 ALLISON A /dev/add02
t 65010733415 Escalation Exhausted (Inform Supervisor) Line 99631665 Event 512036302 ... ...
```

The records above are reasonably obvious once one can understand each records' contents from the single letter code at the start of each. The <System_Name> interpreting data structure is (obviously) a C union, which version of the union being decided by the single letter tag at the start of each record. see mdshsrec.h.

A few brief comments on the examples above:

Message Code	Description of Message
1 m – message	A pager message text
2 c – contact	“Contact” processes messages that need more complex handling. This may be, for example, a record of a fax that was sent as the (previous) SMS message wasn't replied to in, say, 20 minutes.
3 d – dest (Multiple)	Messages can be sent to multiple destinations. This record has a number of destinations ('07') followed by 7 pairs of <ul style="list-style-type: none"> - destination queue (see the queue section of this manual), and - yyymmddHHmmss date-time stamp
4 r – retrieval	A message retrieved from history storage
5 t – contact message	What was actually sent as the result of a “Contact” special handling. In this case it is a message to a Call Centre operator to contact their supervisor - to ensure that the mail gets through.

Table 105: Meaning of Secondary History file fields

First have a look at the following example lines from a secondary history file. These lines appear in the secondary history text file as:

```
{ m 34 1766                                     This is your sample message
{ d 60 1800          01 ∞
{ d 39 2994          02 ka 20021215 232115 em 20021215 232119
{ c 40 1860          S G 20021215 232111 TEST_GRP muoi 340001200
{ t 60 1900          This is your sample message
{ g 0 null_ptr       20021215 232115 D SMS Gate
```

though, by adding their index (byte location from the start of the file) and making a nice little box I have displayed the records as:

1726 { m 34 1766	This is your sample message
------------------	-----------------------------

1766	{ d 60 1800 01 co
2844	{ d 39 2994 02 ka 20021215 232115 em 20021215 232119
1800	{ c 40 1860 S G 20021215 232111 TEST_GRP muoi 340001200
1860	{ t 60 1900 This is your sample message
2994	{ g 0 null_ptr 20021215 232115 D SMS Gate

And they mean:

- First, the italic numbers at the start of each row do not appear in the history file, but I have added them here to indicate the index, or location, of the start of each record in the secondary history file. The primary history records are matched to the appropriate secondary history records by these indexes and the secondary history records are likewise connected to each other with these same indexes.
- Next occurs the start-of-record character, the '{' to ensure that a process reading the secondary history has correctly located itself at the start of a record.
- The first real character is the single letter 'tag', indicating the meaning and format of the remainder of the record.

tag	meaning	description
m	message text	The text of a pager message.
d	Destinations	The queues to which the message is being sent
c	contact	Indicates what type of advanced processing is required. skymds27 handles these messages.
t	text (contact)	The text component of a 'contact' message
g	GSM-SMS reply	Records what carrier (e.g. Optus) delivered each pager message
r		A record that a process and an operator / user (i.e. person) has read some data from the history
v	Voice mail	a record of a voice mail message.

Table 106: Meaning of Secondary History tags

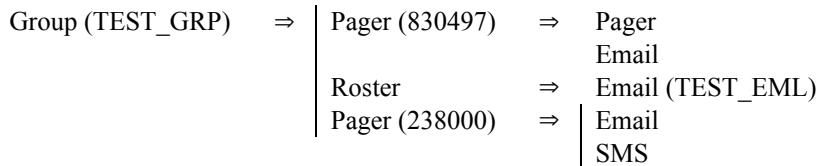
- The next two numbers are the length and index / location of the next secondary history record related to this one. If this record is the last in a set the length of the next record is set to zero and its location to null_ptr.
- Finally, what these data in these records mean

tag	meaning	description
m	message text	The text of a pager message.
d	Destinations	A number indicating the number of destinations , followed by sets of destination queue and maybe a date and time.
c	contact	<ul style="list-style-type: none"> - S or I, indicating the presence or activation (respectively) of a contact record - Next a letter indicating what kind of contact this is, e.g. 'G' for group - Date and time - The name of the contact service (e.g. TEST_GRP) to be activated for this caller. - Finally the operator and the 'event number' this contact has been assigned.
t	text (contact)	The text (only) of a contact message
g	GSM-SMS reply	A date and time stamp, followed by the name of the gateway, and thus the carrier, through which a particular SMS message was delivered.

No ('r') retrievals or ('v') Voice mail messages appear in our example.

16.5 A PRIMARY AND SECONDARY HISTORY FILE EXAMPLE

To give a more detailed explanation of the inner-workings of the history data, let's take an example. The Group paging contact:



produced these following primary (bold) and secondary (bracketed under each primary) history records

	C TEST_GRP 232111 0096250296 D /dev/pts/tm c muoi 340001200 I 40 1726 2954
1726	{ m 34 1766 This is your sample message
1766	{ d 60 1800 01 co
1800	{ c 40 1860 S G 20021215 232111 TEST_GRP muoi 340001200
1860	{ t 60 1900 This is your sample message
1900	{ c 40 1960 S P 20021215 232111 830497 muoi 340001200
1960	{ t 60 2150 This is your sample message
	P 0000830497 232111 0096250296 D /dev/pts/tm C muoi 340001200 I 60 2000 2100
2000	{ c 40 2060 I G 20021215 232111 TEST_GRP muoi 340001200
2060	{ m 50 2100 This is your sample message
2100	{ d 0 null_ptr 02 m5 20021215 232115 er
	C TEST_ROS 232112 0096250296 D /dev/pts/tm C muoi 340001200 I 60 2250 2370
2250	{ c 60 2310 I G 20021215 232111 TEST_GRP muoi 340001200
2310	{ c 40 2370 S r 20021215 232112 TEST_ROS muoi 340001200
2370	{ m 0 null_ptr This is your sample message
	232111 0096250 D
	C 296
	T
	E
	S
	T
	m
	E
	M
	L
2410	{ m 34 2450 This is your sample message
2450	{ d 0 null_ptr 01 em 20021215 232116
	C TEST_EML 232112 0096250296 D /dev/pts/tm C muoi 340001200 I 60 2584 2704
2584	{ c 60 2644 I G 20021215 232111 TEST_GRP muoi 340001200
2644	{ c 40 2704 S L 20021215 232112 TEST_EML muoi 340001200
2704	{ m 0 null_ptr This is your sample message
	P 0000238000 232112 0096250296 D /dev/pts/tm C 70b9e muoi 13 340001200 I 60 2744 2994
2744	{ c 40 2804 I G 20021215 232111 TEST_GRP muoi 340001200
2804	{ m 50 2844 This is your sample message

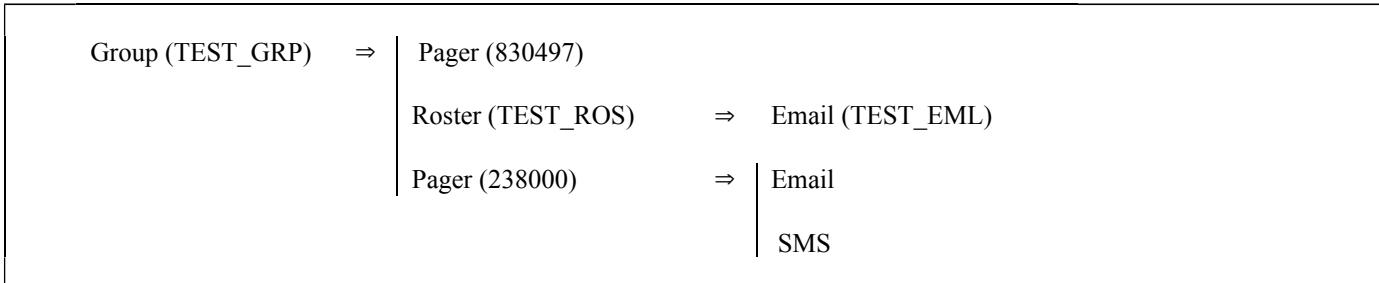
ONLY for personal, private use. Do not retain after use; do **NOT** distribute.

ONLY for use in association with hiring Stephen McGregor as a technical writer.

2844	{ d 39 2994 02 ka 20021215 232115 em 20021215 232119
2894 2954	{ c 40 2954 S P 20021215 232113 238000 muoi 340001200 { t 0 null_ptr This is your sample message
2994	{ g 0 null_ptr 20021215 232115 D SMS Gate

16.6 DETAILED DESCRIPTION OF THE HISTORY RECORD EXAMPLE

To briefly recap our example:



A “group” contact, which sends to two pager messages and a roster, is initiated. The first pager sends a normal pager message, the Roster – another ‘contact’ handled by skymds27 – sends an email, and the second ‘pager’ is forwarded to both an email and to an SMS message. In case you don’t know, a Roster is a <System_Name> service that sends different messages dependent on the date, day of the week, and time, while a Group Message send, as per the example, a single message to multiple destinations.

Now, our history records.

16.6.1.1 The Initial Message’s History

C TEST_GRP 232111 0096250296 D /dev/pts/tm c muoi 340001200 I 40 1726 2954 | skymds20

This first Primary history record is written by skymds20 as it extracts the (Contact) message from the bulk queue. It identifies that the TEST_GRP custom ‘contact’ belonging to the indial 0096200296 has been initiated

Index	Message Content	Description	Generator
1726	m This is your sample message	The message (‘m’) text. This is the primary history record’s text.	skymds20
1766	d 01 co	Message sent to the ‘co’ – contacts queue for skymds27 to read.	skymds20
1800	c S G TEST_GRP muoi 340001200	Our Group (‘G’) contact is being processed (‘S’)	skymds27
1860	t This is your sample message	This Contact’s text	skymds27
1900	c S P 830497 muoi 340001200	A Pager Message is being sent	skymds20
1960	t This is your sample message	The Pager Message’s text	skymds20

16.6.1.2 ‘Spawned’ Pager and Roster message history

P 0000830497 232111 0096250296 D /dev/pts/tm C muoi 340001200 I 60 2000 2100 | skymds20

The primary history record providing an independent record of this pager message spawned by the TEST_GRP group contact.

Index	Message Content	Description	Generator
2000	c I G TEST_GRP muoi 340001200	This pager message was instigated (‘I’) by group (‘G’) TEST_GRP	skymds20
2060	m This is your sample message	The pager message	skymds20
2100	d 02 m5 20021215 232115 er	pager message sends to two message queues, m5 and er	skymds20

The next couple of secondary history records are a continuation of the TEST_GRP group pager message. They identify that the Group has issued a roster

Index	Message Content	Description	Generator
2150	c S r TEST_ROS muoi 340001200	The start of the roster step	skymds27
2210	t This is your sample message	The text of the Roster message	skymds27

16.6.1.3 Detailed Roster Message History

C TEST_ROS 232112 0096250296 D /dev/pts/tm C muoi 340001200 I 60 2250 2370 |skymds27

The Primary History record for the Roster (TEST_ROS) generated by the group contact. The next three secondary history records note the generator (the Group contact TEST_GRP), the start of, and the text of the ‘Roster’ message. The roster, in this case, triggers a TEST_EML email service, as we shall see.

Index	Message Content	Description	Generator
2250	c I G TEST_GRP muoi 340001200	The Generator of this roster	skymds20
2310	c S r TEST_ROS muoi 340001200	Our roster (‘r’) contact is being processed (‘S’)	skymds27
2370	m This is your sample message	The text used by the Roster for its message	skymds27

16.6.1.4 Email contact generated by the Roster and ‘previous’ email

C TEST_EML 232111 0096250296 D /dev/pts/tm c muoi 340001200 I 40 2410 2450 |

The Primary record for the Email contact (TEST_EML) generated by the Roster (above) and its two secondary records (following).

Index	Message Content	Description	Generator
2410	m This is your sample message	The text of the email message	skymds27
2450	d 01 em	Delivered to the email (‘em’) queue	skymds27

The next two secondary records record the email again, this time under the top-level ‘Group’ paging contact that started this all in the first place.

Index	Message Content	Description	Generator
2484	c S L TEST_EML muoi 340001200	Recording that an email (‘L’) step is being activated (‘S’).	skymds27
2544	t This is your sample message	The text of the email message	

16.6.1.5 What is happening here?

C TEST_EML 232112 0096250296 D /dev/pts/tm C muoi 340001200 I 60 2584 2704

Index	Message Content	Description	Generator
2584	c I G TEST_GRP muoi 340001200		
2644	c S L TEST_EML muoi 340001200		
2704	m This is your sample message		

16.6.1.6 Another Email and SMS triggered by the Group's pager message

P 0000238000 232112 0096250296 D /dev/pts/tm C 70b9e muoi 13 340001200 I 60 2744 2994

Index	Message Content	Description	Generator
2744	c I G TEST_GRP muoi 340001200	This Pager message generated by the contact TEST_GRP	skymds27
2804	m This is your sample message	The text of the pager message	skymds27
2844	d 02 ka em 20021215 232119	sent to two queues, Ks ('SMS') and email ('em')	skymds20

There is one further secondary record associated with this pager message primary record, a return receipt for the delivery of the SMS message. There are no further records for the (simple) email sent by from this pager message. The emails sent as contacts (the TEST_EML contacts) in this case *are* individually recorded.

The next two secondary history records record the pager message again, this time against the TEST_GRP group contact that initiated this series of messages.

Index	Message Content	Description	Generator
2894	c S P 238000 muoi 340001200	Recording that a pager ('P') step is being activated ('S').	skymds27
2954	t This is your sample message	The text of the pager message.	skymds27

16.6.1.7 SMS 'return-receipt', detailing which carrier used.

Finally, the history is updated with a record indicating fort the pager message 238000 carrier finally delivered the SMS message. This is indicated by the gate used, in this case the gate is "SMS". This final secondary history record is related to the previous pager message primary history record.

Index	Message Content	Description	Generator
2994	g D SMS Gate	Which delivery system was SMS message was sent over	skymds03

HICCYMMMD – PRIMARY HISTORY DATABASE TABLES

Primary history files , which exist as C-ISAM database tables, are named after the date on which they were created, e.g. hi20010928 is the primary history table for the 29-Sept-2001. The primary history contains information *about* messages. See the secondary history text file for the message itself

```
*****
* MDSHIREC.H      HISTORY LOGGING FILE
* _____
* *
*   FILE      : hiccyymmd
* RECORD LENGTH: 120 (including LF)
* DATE MODIFIED: 23/11/98 14:48
* INDEX DETAILS: 1 NODUPS 0/ 1 hist_type
*                 1/10 pager_contact_id
*                 11/ 6 time_recv
*                 17/10 line_num
*                 27/ 1 machine_id
* 2 NODUPS 0/ 1 hist_type
*                 17/10 line_num
*                 1/10 pager_contact_id
*                 11/ 6 time_recv
*                 27/ 1 machine_id
* COPYRIGHT : <Company Name> Telecommunications (C) 1995.
*             All rights reserved.
* NOTES     : This record defines the 'logical' history file,
*             encompassing all the daily history files currently online.
*             The 'real' history record doesn't contain "date_recv".
*****
#define HIST_KEY_1 '1'
#define HIST_KEY_2 '2'
#define HIST_KEY_3 '3'
#define DUPLICATE_SUCCESS 2

#define KEY_HI_1 "00-1,1-10,11-6,17-10,27-1"
#define KEY_HI_2 "00-1,17-10,1-10,11-6,27-1"
#define KEY_HI_3 "D17-10,11-6"
```

```
typedef struct {
    CC_DATE      date_recv; /*BEWARE*/ /* -/ 8 Date this message was recvd */
    char         id_type;  /* 0/ 1 ID type
                           /* 'P' Pager
                           /* 'C' Contact */
    PAGNUM       pager_contact_id; /* 1/10 Pager number or contact id */
    TIME         time_recv; /* 11/ 6 Time this message was recvd */
    LINE_NUM     line_num;  /* 17/10 Line number call originated */
    char         machine_id; /* 27/ 1 Code of the originating machn */
    PORT         port;      /* 28/16 Port message was entered from */
    char         tran_type; /* 44/ 1 Transaction type:- */
                           /* ' ' - ordinary message */
                           /* 'B' - Booked Call Message */
                           /* 'C' - Queued by Cont Clr Sys */
                           /* 'c' - Contact selected */
                           /* 'D' - Delayed Delivery */
                           /* 'X' - Extra (repeat) msg */
                           /* 'b' - Batch Fax */
                           /* 'd' - Dummy */
                           /* 'l' - Email Report */
                           /* 's' - Start follow me */
                           /* 'e' - End follow me */
                           /* 'M' - GL3000 Mail Box */
                           /* 'G' - GL3000 message */
    PAGNUM       orig_id;   /* 45/10 alt pager or smsc msg id */
    OPERATOR     operator;  /* 55/ 8 Operator who entered the msg */
    char         sequ_filler; /* 63/ 1 Filler */
    FORM2        msg_sequ_nbr; /* 64/ 2 where applicable */
    char         event_filler; /* 66/ 1 Filler */
    EVENT_NUM    event_num;  /* 67/ 9 Event number */
    char         page_initiator; /* 76/ 1 Where message was originated */
                               /* 'O' Operator */
                               /* 'E' External */
                               /* 'I' Internal */
    TRAN_LEN     next_tran_len; /* 77/ 4 Length of first transaction */
    HSEC_PIR    first_hsec_ptr; /* 81/ 9 Pointer to first hsec record */
    HSEC_PIR    last_hsec_ptr; /* 90/ 9 Pointer to last hsec record */
    char         filler[20]; /* 99/ 20 */
    char         lf;          /* 119/ 1 */
} HIREC;
#define HIRECSIZE sizeof(HIREC)
```

SECONDARY HISTORY DATA LAYOUT

The secondary history is stored as text files, named after the date on which they were created, e.g. hs021123.txt is the secondary history file for <date removed>. The secondary history contains the information about the message itself, c.f. information relating to the messsage's transmission and background in the primary history.

```
*****
* MDSHREC.H      HISTORY SECONDARY STORAGE FILE
* _____
*           *
*   FILE      : hsymrdd
*   RECORD LENGTH: 976 (maximum size - including LF)
*   DATE MODIFIED: 29/12/98 10:00
*   INDEX DETAILS: Direct access file
*   COPYRIGHT  : <Company_Name> Telecommunications (C) 1995.
*               All rights reserved.
*****
```

```
typedef struct {
    DEST          dest;
    CC_DATE_TIME date_time_sent;
} DEST_DATE_TIME;
```

```
#define HS_CO_MESSAGE_TYPE      'm'
#define HS_CO_RETRIEVAL_TYPE     'r'
#define HS_CO_DEST_SENT_TYPE     'd'
#define HS_CO_CONTACT_STEP_TYPE  'c'
#define HS_CO_CONTACT_MSG_TYPE   't'
#define HS_CO_SMS_FINAL_MSG_TYPE 'g'
#define HS_CO_GL_VOICE_MAIL_TYPE 'v'

#define HS_HC_NORMAL_STEP        'S'
#define HS_HC_INIT_STEP          'I'
```

```
typedef union {
    struct { /* HEADER COMMON TO ALL TRANSACTION TYPES */
        char magic_nbr;           /* 0/ 1 Integrity verification = '{' */
        char ttran_type;          /* 1/ 1 Transaction type:- */
        /* 'm' - message.          */
        /* 'r' - retrieval.        */
        /* 'd' - dest delivery.   */
        /* 'c' - contact step.    */
        /* 't' - contact message. */
        /* 'g' - GSM-SMS final state */
        /* 'v' - GL Voice mail   */
    } co;
    #define HS_PREFIX_SIZE 15      /** BEWARE make same as size above **/
```

```
    struct { /* MESSAGE TRANSACTION */
        char common[HS_PREFIX_SIZE];/* 0/ 15 Definition common to all */
        MESSAGE message;           /* 15/960 pager message */
        char lf;                   /* 975/ 1 */
    } hm;
```

```
    struct { /* MESSAGE RETRIEVAL TRANSACTION */
        char common[HS_PREFIX_SIZE];/* 0/ 15 Definition common to all */
        CC_DATE date_retrieved;    /* 15/ 8 Date of retrieval */
        TIME   time_retrieved;     /* 23/ 6 Time of retrieval */
        LINE_NUM line_num;         /* 29/ 10 Line num retrieved on */
        OPERATOR operator;        /* 39/ 8 Operator did the retrieval */
        char machine_id;          /* 47/ 1 machine done on */
        PORT   port;              /* 48/ 16 port done on */
        char lf;                  /* 64/ 1 */
    } hr;
```

```
    struct { /* DESTINATIONS 'TO BE' or 'HAVE BEEN' DELIVERED TRANSACTION */
        char common[HS_PREFIX_SIZE]; /* 0/ 15 Defn common to all */
        FORM2 nbr_dests;           /* 15/ 2 nbr entries in next fld */
        DEST_DATE_TIME sent_dest[TOT_DESTS]; /* 17/960 dest and time to enc */
        char lf;                   /* 977/ 1 */
    } hd;
```

```

struct { /* CONTACT CLEARING DETAILS */
    char common[HS_PREFIX_SIZE]; /* 0/ 15 Definition common to all */
    char step_type; /* 15/ 1 'I' - Initiator Identifier */
    /* 'S' - Normal Step */
    char sub_tran_type; /* 15/ 1 'A' - Advisory */
    /* 'P' - Page from contact */
    /* 'r' - Roster initiated */
    /* 'E' - Escalation initiated */
    /* 'M' - Reminder initiated */
    /* 'G' - Group initiated */
    /* 'T' - TMS */
    /* 'F' - FAX */
    /* 'f' - ERM */
    /* 'p' - Phone */
    /* 'm' - Manual Reminder */
    /* 'a' - Escalation ACKnowledge*/
    /* 't' - TMS ACKnowledge */
    /* 'b' - Fax Report */
    /* 'S' - Supervisor reminder */
    /* 'C' - Reminder ACTIONed */
    /* 'N' - Reminder N/Answer */
    /* 'e' - Reminder ENGaged */
    /* 'L' - Email Message */
    /* 'c' - Call Patch CPA */
    /* 'x' - Text File */
    CC_DATE date_processed; /* 16/ 8 */
    TIME time_processed; /* 24/ 6 */
    CONTACT_ID contact; /* 30/ 10 */
    OPERATOR operator; /* 40/ 8 */
    char event_filler; /* 48/ 1 */
    EVENT_NUM event_num; /* 49/ 9 */
    char lf; /* 58/ 1 */
} hc;

struct { /* CONTACT MESSAGE */
    char common[HS_PREFIX_SIZE]; /* 0/ 15 Definition common to all */
    MESSAGE message; /* 15/960 pager message */
    char lf; /* 975/ 1 */
} ht;

```

```

struct { /* SMSC FINAL STATE TRANSACTION */
    char common[HS_PREFIX_SIZE]; /* 0/ 15 Definition common to all */
    CC_DATE date_finalised; /* 15/ 8 */
    TIME time_finalised; /* 23/ 6 */
    char deliv_status; /* 29/ 1 Final message status */
    /* 'D' - Delivered */
    /* 'd' - Deleted */
    /* 'E' - Expired */
    /* 'e' - En-route */
    /* 'I' - Invalid */
    SMSC_ID smsc_id; /* 30/ 8 */
    char lf; /* 38/ 1 */
} hg;

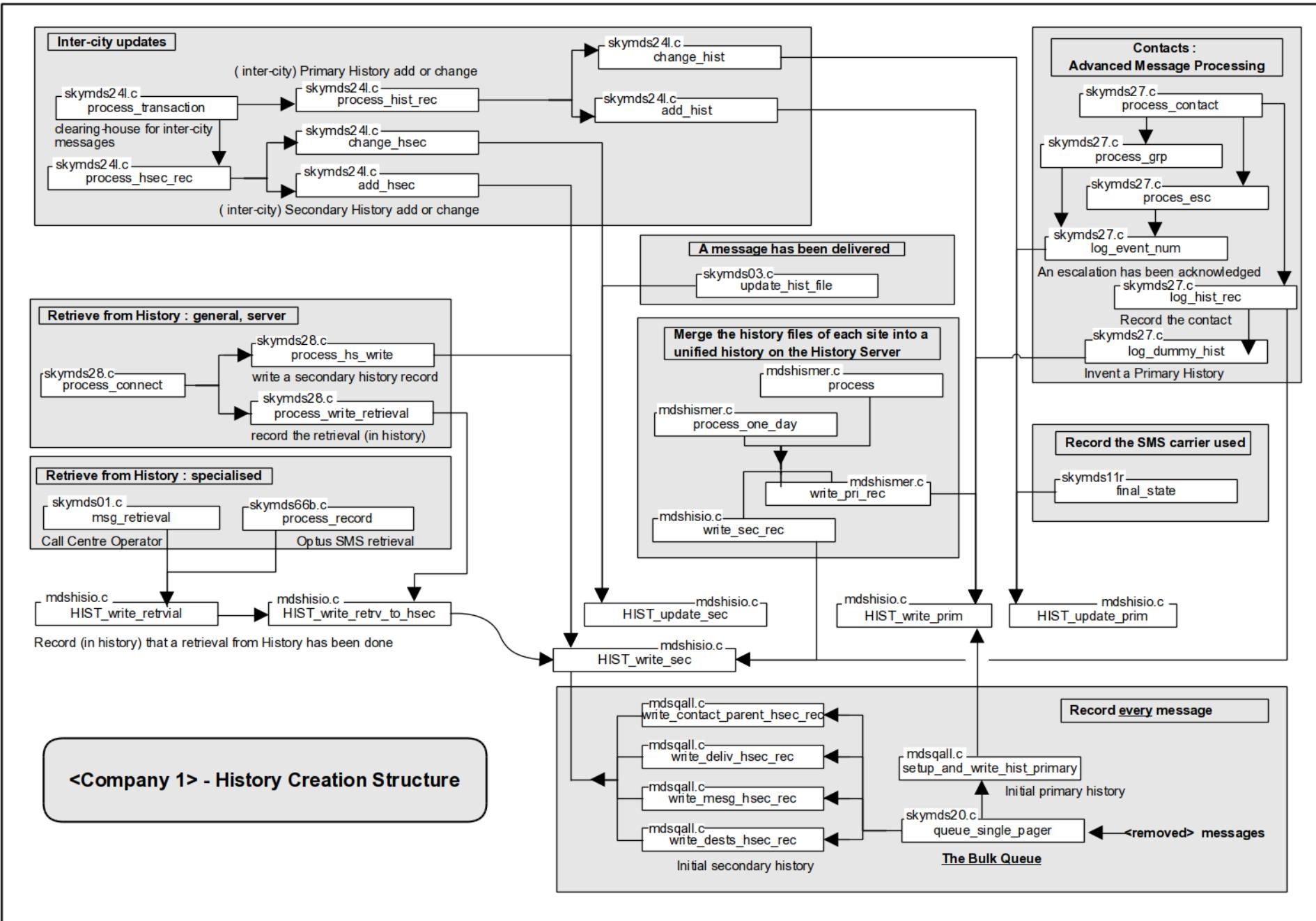
char the_lot[978];

} HSREC;

#define HSRECSIZE sizeof(HSREC)

#define HSEC_CO_SIZE sizeof(hs.co)
#define HSEC_FM_SIZE sizeof(hs.fm)
#define HSEC_HR_SIZE sizeof(hs.hr)
#define HSEC_HD_SIZE sizeof(hs.hd)
#define HSEC_HC_SIZE sizeof(hs.hc)
#define HSEC_HI_SIZE sizeof(hs.ht)
#define HSEC_HG_SIZE sizeof(hs.hg)

```



17 CDR - Call Detail Record

CDRs are the raw billing-related data within <System_Name>. Call Detail Records are written by skymds01 - the Call Centre interface program - and often by skymds20 - the program that extracts messages from the Bulk Queue for allocation to distribution queues. skymds01 writes CDRs for all types of calls, including those that are incomplete, while skymds20 adds CDRs as part of its queue processing.

On each local machine, all CDR data is written into /var/mds/data/cdr into text files named CDRMccyyymmdd (e.g. CDRM20020720 - M and O for Melbourne, S for Sydney ...). At the end of each day these are copied to /var/mds/data/cdr in Adelaide (server lnka). From here they are combined (into CDRALLyyyymmdd) from whence a utility transfers this large daily collection of billing information to the Melbourne LMS server for uploading to an Oracle database for subsequent financial processing. From 250,000 to 300,000 CDR billing records are transferred each day.

The South Australian Government has their own <System_Name> server and, as they have unique billing needs, a separate collection of CDR records are compiled for this organisation. These are only to be found on the lnkg server, in the (unique) /var/mds/data2/ directory.

The CDR file format is defined in mdscdrec.h, with mdsacrec.h defining an extended version that wraps and extends a standard CDR record. mdscdr.h is a long list of definitions, constants and settings for the CDR process.

17.1.1 CDR FILE STRUCTURE

As noted, the /var/mds/data/cdr/CDRMccyyymmdd is a local CDR billing file whose structure is defined in mdscdrec.h. Following is the detail of one CDR line, followed by a full CDR Matrix, listing all of the current CDR line meanings.

An example of a CDR line:

20020720001700|00|00|300984|Q|2|05|0003119999|3||L|0000238031|m5|0000079|5|1|35123900|N|L|92090000|/dev/mla31|L|L|~

This line can be interpreted:

Field Value	Meaning
20020720001700	Date, time
00	Transaction Type
00	Sub Type
300984	Operator ID
0	What machine (O is Melbourne #2)
2	External
05	“CPU” – external computer
0003119999	Indial Number
3	Indial Type. 3 means remote access
[N/A]	2 indicates personalised Service
L	Subscriber type: ‘L’ means <Company_Name>
0000238031	Pager number
[N/A]	Contact (special handling) ID
m5	RF frequency or email address
0000079	message length
5	Which Frequency. 5 means 149.6125 MHz
1	Transmission rate. 1 means 1200 baud
35123900	Pager Client Number
N	Priority? (No)
L	Subscriber Type (<Company_Name>)
92090000	Where the call was made from
/dev/mla31	The Port
L	Subscriber Type
L	Subscriber sub-type

Table 107: Meaning of CDR file fields

See over for a more detailed CDR Matrix.

17.1.2 CDR FILE STRUCTURE MATRIX

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	
Y	00 Paging	00 Alpha	Y	Y	Y	Y	Y	Y	Y	Y	N	RF dest	Length	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
Y		01 Numeric	Y	Y	Y	Y	Y	Y	Y	Y	N	RF dest	Length	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
Y		02 Tone	Y	Y	Y	Y	Y	Y	Y	Y	N	RF dest	Length	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	
Y	01 Incalls	01 Inactive	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	Y	Y	Y	Y	N	N
Y		02 Invalid	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	Y	N	N	N
Y		03 Call Duration	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	No of Seconds	N	N	N	N	N	Y	Y	Y	N	N	N
Y	02 Faxing	04 On Line Fax Pager	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Fax	Length	N	N	Y	Y	Y	Y	Y	Y	Y	N	N
Y		05 On Line Fax Indial	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Fax	Length	N	N	N	N	N	Y	Y	Y	Y	N	N
Y		06 Batch Fax Pager	N	Y	N	N	N	N	N	N	Y	N	Fax	Num Pages	N	N	Y	Y	Y	N	N	N	N	Y	N
Y		07 Batch Fax Indial	N	Y	N	N	Y	Y	Y	Y	N	N	Fax	Num Pages	N	N	N	N	N	Y	N	Y	N	N	N
Y		08 Customised Fax Reports	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Fax	Num Pages	N	N	N	N	N	Y	Y	Y	N	N	N
Y	03 Email	04 On Line Email Pager	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	Email	Length	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
Y		05 On Line Email Indial	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	Email	Length	N	N	N	N	N	Y	Y	Y	Y	N	N
Y		06 Batch Email Pager	N	Y	N	N	N	N	N	N	Y	N	Email	Num Pages	N	N	Y	Y	Y	N	N	N	N	Y	Y
Y		07 Batch Email Indial	N	Y	N	N	Y	Y	Y	Y	N	N	Email	Num Pages	N	N	N	N	N	Y	N	Y	N	N	N
Y	04 Message To TMS	00 TMS Message	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	Y	N	N
Y	05 Message To Text	00 TEXT Message	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	Y	N	N
Y	06 Manual Reminder	00 Manual Reminder	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	Y	N	N
Y		01 Booked Page Add	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	N	N	N	N	N	N	Y	Y	Y	N	N	N
Y		02 Booked Page Change	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	N	N	N	N	N	N	Y	Y	Y	N	N	N
Y		03 Booked Page Delete	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	N	N	N	N	N	N	Y	Y	Y	N	N	N
Y	07 Contact Processed	00 Advisory	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	N	N	N
Y		01 Call Patch	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	N	N	N
Y		02 Email	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	N	N	N
Y		03 Escalation	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	N	N	N
Y		04 Fax	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	N	N	N
Y		05 Form	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	N	N	N
Y		06 Group	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	N	N	N
Y		07 Manual Reminder	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	N	N	N
Y		08 Pager Contact	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	N	N	N
Y		09 Reminder	Y	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Length	N	N	N	N	N	Y	Y	Y	N	N	N

Table 108: More detailed CDR Matrix

```

*****CALL DETAILS MASTER FILE*****
* MOSDREC.H      CALL DETAILS MASTER FILE          *
* FILE           : mdsdrec                         *
* RECORD LENGTH: 281 (including LF)                 *
* DATE MODIFIED: 24/09/02   11:00                   *
* COPYRIGHT       : <Company Name> Telecommunications (C) 1995.    *
*                  All rights reserved.             *
*****/




typedef struct {
    CC_DATE_TIME    date_time;        /* 0/14 Century/Date/Time      */
    CDR_TRANSACTION transaction_type; /* 14/ 2 Transaction type     */
    CDR_TRANS_SUB   trans_sub_type;  /* 16/ 2 Transaction Sub type */
    OPERATOR        operator;        /* 18/ 8 Operator entered message */
    char            machine_id;     /* 26/ 1 Machine Id          */
    char            source;          /* 27/ 1 0 = Internal         */
                                    /* 1 = Operator               */
                                    /* 2 = External                */
    CDR_SUB_SOURCE  sub_source;     /* 28/ 2 00 = System Alarm - Int */
                                    /* 01 = Paging Terminal - I/E/O */
                                    /* 02 = Activations (Wang) -Int */
                                    /* 03 = Internet - Ext        */
                                    /* 04 = Indial Box - Ext      */
                                    /* 05 = CEU - Ext             */
                                    /* 06 = Booked Call - Int/Opr */
                                    /* 07 = Follow Me - Int/Opr  */
                                    /* 08 = IVR - Ext             */
                                    /* 09 = Repeat Page - I/O/E   */
                                    /* 10 = Delayed Page - I/O/E  */
                                    /* 11 = SAGRN (GL3000) - Ext */
    CDR_INDIAL_NUM  indial_number;  /* 30/20 Line Number          */
    char            indial_type;    /* 50/ 1 Indial Type          */
                                    /* 0 = General Paging          */
                                    /* 1 = Tone/Numeric            */
                                    /* 2 = Personalised Service   */
                                    /* 3 = Remote Access           */
                                    /* 4 = Mailbox (GL3000)        */
    CDR_SERVICE_TYPE service_type; /* 51/ 2 For Personalised Services */
                                    /* ie. CAS, EMP, LIM, etc..   */
    char            li_subscriber_type; /* 53/ 1 Identifies Subscriber type */
                                    /* G = SA Government           */
                                    /* L = <Company Name>          */
                                    /* S = SurePage                */
                                    /* T = Telstra                 */
    CDR_PAG_MOB_NUM pag_mob_nbr;  /* 54/20 Pager/Mobile Number   */
    BULK_CONTACTID contact_id;   /* 74/20 Contact Id           */
                                    /* For bulk batch id it is 20 chars */
                                    /* For contact_id it's 10 chars only */
    CDR_DEST_DELIVERY dest_delivery; /* 94/80 RF Dest              */
                                    /* Phone/Fax/Modem Number      */
                                    /* Email Address                */
    CDR_VOLUME      volume;        /* 174/ 7 Msg Length, A4 pages, */
                                    /* File Size, Call Duration    */
}

```

```

char          frequency;      /*181/ 1 Frequency of the pager */
                            /* 1 = 149.8875 Mhz           */
                            /* 2 = 149.8375 Mhz           */
                            /* 3 = 149.7875 Mhz           */
                            /* 4 = 149.9625 Mhz           */
                            /* 5 = 149.6125 Mhz           */
                            /* 6 = 148.8125 Mhz           */
char          baud_rate;      /*182/ 1 0 = 512 Baud          */
                            /* 1 = 1200 Baud               */
                            /* 2 = FLEX 1600/2400 Baud     */
                            /* 9 = Undefined               */
CLIENT_NUM    pa_client_num;  /*183/ 8 Pager Client Number   */
char          pager_priority; /*191/ 1 Priority              */
char          pa_subscriber_type; /*192/ 1 Pager Subscriber Type */
CLIENT_NUM    li_client_num;  /*193/ 8 Line Client Number    */
PORT          port;          /*201/16 Port                  */
char          li_subscriber_sub_type; /*217/ 1 Line Subscriber Sub-Type */
char          pa_subscriber_sub_type; /*218/ 1 Pager Subscriber Sub-Type */
SOURCE_ADDRESS source_addr; /*219/60 Source address        */
char          end_of_rec;    /*279/ 1 End Of Record Specifier */
char          lf;            /*280/ 1                         */
} CDRREC;

```

```

#define CDRRECSIZE sizeof(CDRREC)
#define CDR_DIR    "cdr/"
*****ADVISORY FILE*****
* MOSADREC.H      ADVISORY FILE          *
* FILE           : mdsadvis                         *
* MAINENANCE: <SYSTEM NAME>                      *
* RECORD LENGTH: 845 (including LF)                 *
* DATE MODIFIED: 11/11/98 13:10                   *
* INDEX DETAILS: NODUPS 0/10 line_num             *
*                           10/10 contact_id          *
* COPYRIGHT       : <Company Name> Telecommunications (C) 1995.    *
*                  All rights reserved.             *
*****KEY AD 1 "00-10,10-10"
typedef struct {
    LINE_NUM      line_num;        /* 0/10 line number (DNIS)      */
    CONTACT_ID    contact_id;    /* 10/10 contact identifier    */
    FORM2         nbr_lines;     /* 20/ 2 number of lines of text */
    ADV_LINE      adv_line[10];  /* 22/800                       */
    OPERATOR      mod_operator;  /* 822/ 8                         */
    CC_DATE_TIME  mod_time;     /* 830/14                         */
    char          lf;            /* 844/ 1                         */
} ADREC;
#define ADRECSIZE sizeof(ADREC)
#define MDSADVVIS  "mdsadvis"

```

18 skymds00j - Client Maintenance

skymds00j implements the <System_Name> Client Maintenance window available from the base <System_Name> menu (skymds00).

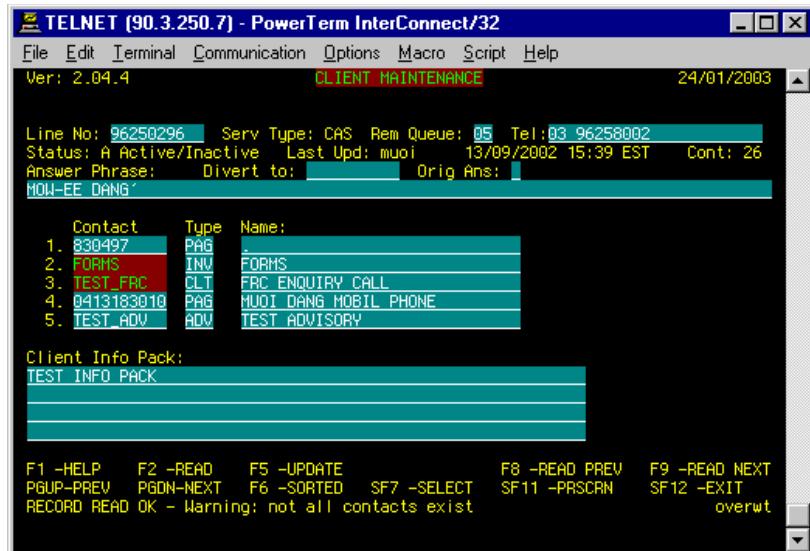
coded in: skymds00j.c

called from (code): the main_menu() function in skymds00.c

called from (window): Selecting 'Client' from the <System_Name> root menu window.

raison d' etre: Provide an interface to the management of a client's information, focusing on the contacts (Advanced message processing services) they have set up.

screen:



database tables:
all tables

18.1 PROGRAM & SCREEN INITIALISATION

skymds00j's main() routine calls two functions:

- initialise(), which performs the usual <System_Name> program start-up,
- file_edit(), which maintains control until the user closes the Client Maintenance window and skymds00j exits

18.1.1 SCREEN CONSTRUCTION

The screen is defined globally and constructed in the first part of file_edit(), like this:

- Screen Allocation: first, two calls to create_popup_win() allocate the parts of the screen, the status line and everything else.
- input fields: Next clear_screen() empties out all the input fields in (a copy of) the global screen structure, calling wrefresh() to display these fields
- text headings: file_screen() writes in all the text headings
- contact info: display_record() writes the contact information into the contact info box.

18.2 FUNCTION: FILE_EDIT

file_edit() is the central switchboard function to skymds00j. Handling the key-presses from the user, file_edit() either:

- calls edit_scrn() to handle the normal navigation around, and editing of, the screen, or
- calls one of a number of specialised work functions to perform special, important tasks, including saving the updates made during calls to edit_scrn()

The main use of skymds00j is as an entry point to the management of clients' contacts (advanced message-processing services).

18.2.1.1 General usage keys and functions

Key	Function called	Description
F2	file_search()	reads the line (mdsline) for general information such, as the answer phrase, and calls get_contacts() to get the contacts set up for this indial.
F5	file_update()	saves any changes the user has made. After checking security file_update() calls validate_screen() and update_records() to check and write the changes.
F8	file_prev()	Moves to the previous client in the mdsline table. Not much use in this context.
F9	file_next()	moves to the next client in the mdsclien table.
F10	screen_ruler()	super-imposes a 'ruler' over the contact name field.

18.2.1.2 Contact Management Functions

Key	Function called	Description
shift-F7	sort_contacts() select_contact()	Reorders the contacts alphabetically by name If the cursor is positioned on one of the contacts in the contact list, shift-F7 calls the contact set-up program.
page-down	next_page()	scroll down the list of contacts
page-up	prev_page()	move up the list of contacts.

To add a new contact:

1. move to the middle column of the contact list in the middle of the screen
2. Enter a contact identifier, a 3 digit contact code, and a contact name. The contact name is for display use only, while the 3 digit code must be one of the 14 codes appearing in the list appearing in the following discussion of the select_contact() function.
3. Hit shift-F7, calling select_contact(), to display the appropriate screen for the type of contact code you entered in step 2.

18.2.1.3 Hidden, un-noted, functions

Key	Function called	Description
shift-F2	client_stats()	calls client_stats() to display the number of calls processed (over the previous week and year) for this indial
shift-F3	pager_stats()	After prompting for a pager number, displays the same information as does client_stats()

18.3 FUNCTION: SELECT_CONTACT

`select_contact()` enables the user to edit a client's contact (advanced processing services) details by transferring control to a function managing the selected contact type.

Once a user positions their cursor on a row in the client management screen's contact table and pushes shift-F7, `select_contact()` reads the three digit contact-type code and calls the appropriate handler. As detailed in the following table, each contact maintenance handler appears in a separate source code file `mds0j????.c` (e.g. `mds0jros.c`) and is named appropriately (e.g. `ros_roster_details()`).

Code	Function Called	coded in:	Description
ADV	<code>adv_advisory_details()</code>	<code>mds0jadv.c</code>	Read a message (to the caller) and collect a reply
CLT, INV	<code>file_edit() [recursively]</code>	<code>(skymds00j.c)</code>	recursively add more contacts. End up back here ...
CPA	<code>cpa_call_patch_details()</code>	<code>mds0jcpa.c</code>	connect the caller to an external phone number.
EML	<code>eml_email_details()</code>	<code>mds0jeml.c</code>	send an email
ESC	<code>esc_escalation_details()</code>	<code>mds0jesc.c</code>	sent repeatedly until an acknowledgement
FAX	<code>fax_fax_details()</code>	<code>mds0jfax.c</code>	send a fax
FRM	<code>frm_form_details()</code>	<code>mds0jfrm.c</code>	Forms, customised interfaces.
GRP	<code>grp_group_details()</code>	<code>mds0jgrp.c</code>	one message sent to many destinations
REM	<code>rem_reminder_details()</code>	<code>mds0jrem.m</code>	messages to the Call Centre Operators
ROS	<code>ros_roster_details()</code>	<code>mds0jros.c</code>	time and date dependent delivery behaviours
SCH	<code>sch_search_par_details()</code>	<code>mds0jsch.c</code>	find a message destination by searching
TMS	<code>tms_tms_details()</code>	<code>mds0jtms.c</code>	recording messages for delivery later.
TXT	<code>text_txt_details()</code>	<code>mds0jtxt.c</code>	like tms, but stored in a file and emailed

After the function called by `select_contact()` returns, `select_contact` calls `recheck_contacts()` and `display_record()` to check and display any visible changes. It then completes, returning control to the key-reading loop in `file_edit()`.

18.4 GENERAL FORM OF CONTACT MAINTENANCE FUNCTIONS

The contact management functions are very similar in form, which I describe here.

For the contact of type *contact* with a three letter abbreviation *CCC*, what happens is:

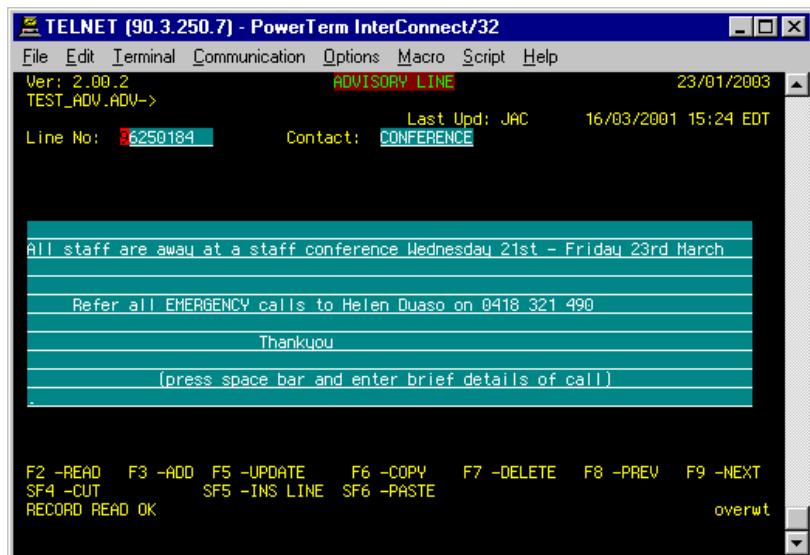
1. When the user positions the cursor on a contact display line with code *CCC* and pushes shift-F7, the function `ccc_contact_details()` is called. This function is coded in the file `mds0jccc.c`
2. `ccc_contact_details()` writes all of the static text to the screen, making a sub-window for the information lines at the top of the screen and the key-instructions at the bottom
3. It then calls `ccc_process_contact()` to manage the user interaction. One of these two functions, usually this one (`ccc_process_contact()`) calls `ccc_set_screen_array()` to place the editable text-boxes on the screen, and `ccc_file_read()` to write any existing database info into these text boxes.
4. `ccc_process_contact()` then starts an infinite loop, using `edit_sern()` to read the user's key input, passing the function keys back to `ccc_process_contact()`. These keys trigger, in a consistent pattern across contacts, various contact editing functions.

Note that the layout of the screen and the available function-key functionality completely determines what contact editing logic is available. No additional business rules are present.

18.5 ADV ADVISED DETAILS: ADVISORY MAINTENANCE

coded in: mds0jadv.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, 'ADV' contact line selected, shift-F7 pushed
raison d' etre: set-up and maintain an advisory for the client. An advisory is a message to be read to the caller with optional data collection. Unlike a temporary message (q.v.) it may be a permanent service

screen:



database tables:

mdsadv
mdsline

18.5.1 PROGRAM & SCREEN INITIALISATION

- first adv_advisory_details() calls adv_set_screen_array() to construct the set of 12 advisory lines in the middle of the advisory screen.
- next adv_advisory_details() constructs the remainder of the screen.

Each couple of lines, the header, the 'last Upd and Line No' the 'F2 -READ ...' blocks are each constructed as separate windows. Once the screen is constructed control passes to adv_process_advisory() to implement the logic of advisory maintenance, which is constrained (only) by the structure of the screen layout and the functionality bound to special keyboard keys.

18.5.2 ADVISORY MAINTENANCE: FUNCTION: ADV_PROCESS_ADVISORY

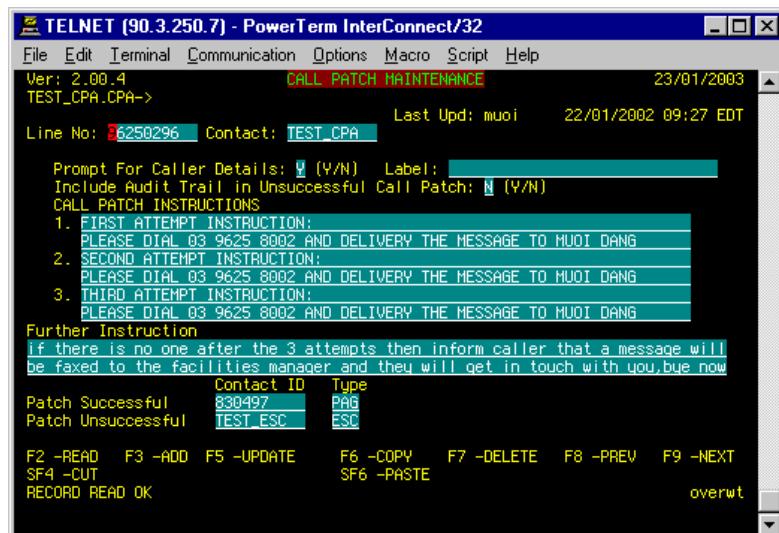
First adv_file_read() is called to read existing advisory details from the mdsadv database table. Next key presses are read by edit_scrn(), the following keys being passed back (here) to be handled.

Key	Function Called	Description
F2	adv_file_read()	read the mdsadv database table (again).
F3	adv_file_add()	write a new advisory record to the mdsadv database table.
shift-F4	adv_del_line()	delete a single advisory line. F7 deletes all the advisory lines
F5	adv_file_update()	write changes back to the same advisory record.
shift-F5	adv_line_insert()	insert a line into the advisory
F6	adv_copy()	take a copy of an Advisory record (for pasting as a new record)
shift-F6	adv_paste()	paste the previously taken copy as a new advisory record.
F7	adv_delete()	delete all the advisory records
shift-F7		if the insertion-point is on the line number, return to the client maintenance screen.
F8,F9	adv_prev(), adv_next()	move to the previous or next advisory in the database. Why??

18.6 CPA_CALL_PATCH_DETAILS

coded in: mds0jcpa.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, 'CPA' contact line selected, shift-F7 pushed
raison d' etre: Setup and maintain 'call patch' instructions. A call patch is when a Call Centre Operator is instructed to connect the caller to an external phone number, as per a telephone exchange operator

screen:



database tables:

mdscallp
mdsline
mdspager

18.6.1 SCREEN INITIALISATION

A pattern is starting to emerge. The call-patch maintenance screen is constructed via the following steps:

- Sub windows (the couple of detail lines at the top of the screen and the key-press instructions at the bottom) and the set text are constructed in cpa_call_patch_details()
- cpa_call_patch_details() passes control to cpa_process_call_patch()
- cpa_process_call_patch() uses cpa_set_screen_array() to display the 17 editable text-boxes on the screen
- cpa_file_read() writes any existing call-patch instructions onto the screen and updates the display

The display is now complete, as per the above example.

18.6.2 CALL - PATCH MAINTENANCE: FUNCTION: CPA_PROCESS_CALL_PATCH

The first two actions of the cpa_process_call_patch() function, the calls to cpa_set_screen_array() and cpa_file_read(), have been explained above. After these two calls, cpa_process_call_patch() enters an infinite loop, using edit_scrn() to process user key-strokes. The keys listed below perform special function on the call-patches listed.

Key	Function Called	Description
F2	cpa_file_read()	read the call patch database table, mdscallp, again
F3	cpa_file_add()	write a new call-patch record to the mdscallp database table.
shift-F4	cpa_delete_line()	delete a line in the call patch instructions
F5	cpa_file_update()	write changes back to the same call-patch database table record.
F6	cpa_copy_line()	copy a line for later pasting.
shift-F6	cpa_paste()	paste a previously copied line
F7	cpa_file_delete()	delete all the call-patch details for this client
shift-F7	if on the client's indial, then return to the client maintenance screen	
F8	cpa_file_prev()	Move to the previous client's call-patch record
F9	cpa_file_next()	Move to the next client's call-patch record
ESC, F12	return to the client maintenance screen	

18.7 EML_EMAIL_DETAILS

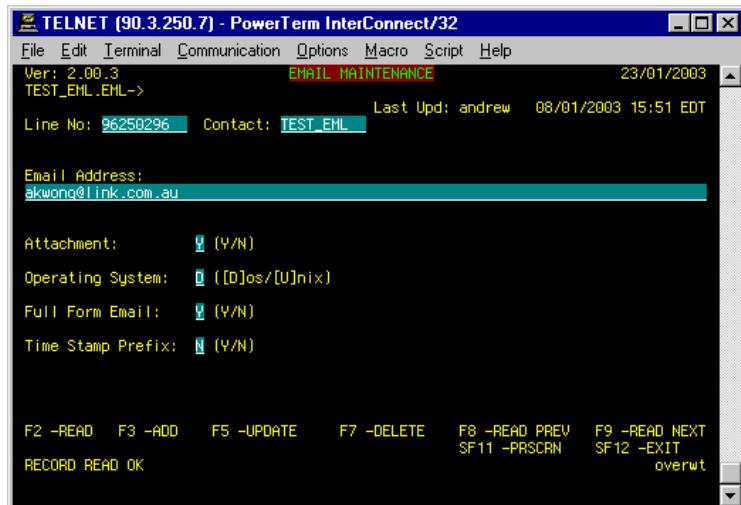
coded in: mds0jeml.c

called from (code): select_contact() in skymds00j.c

called from (window): client maintenance window, 'EML' contact line selected, shift-F7 pushed

raison d' etre: Establish and maintain email-sending contacts.

screen:



database tables:

mdsemail

18.7.1 SCREEN INITIALISATION

The email maintenance screen is constructed via the following steps:

- Sub windows (the couple of detail lines at the top of the screen and the key-press instructions at the bottom) and the set text are constructed in eml_email_details()
- eml_email_details() passes control to eml_process_email()
- eml_process_email() uses eml_set_screen_array() to display the editable text-boxes, 7 of them
- eml_file_read() writes any existing details onto the screen, and updates the display

The display is now complete, as per the above example.

18.7.2 EMAIL MAINTENANCE: FUNCTION: EML_PROCESS_EMAIL

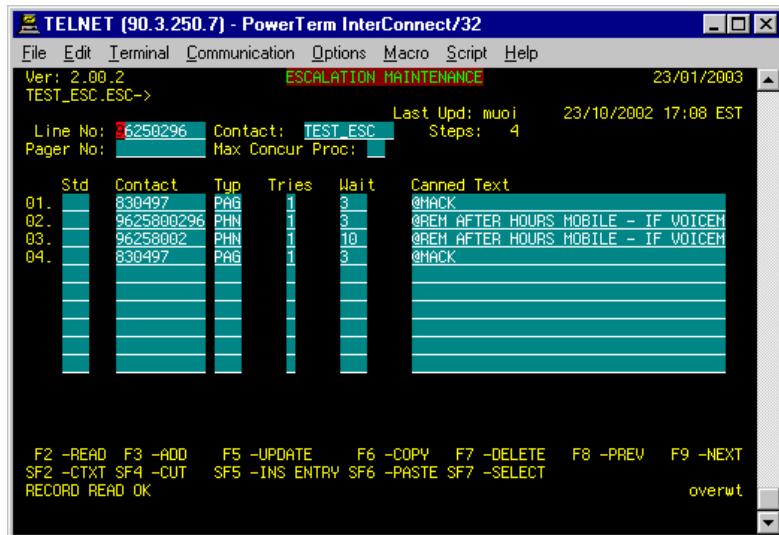
As just described, eml_process_email() starts by calling eml_set_screen_array() and eml_file_read() to complete the preparation of the email maintenance screen. Now eml_process_email() enters an infinite loop, using edit_scm() to process user key-strokes, returning the keys listed in the table to activate special email maintenance tasks. Note that email maintenance is constrained (only) by the screen layout and the functionality bound to special keyboard keys.

Key	Function Called	Description
F2	eml_file_read()	read the email database table, mdsemail, again
F3	eml_file_add()	write a new email record to the mdsemail database table.
F5	eml_file_update()	write changes back to the same email database table record.
F7	eml_file_delete()	delete all the email details for this client
shift-F7	if on the client's indial, then return to the client maintenance screen	
F8	eml_file_prev()	Move to the previous client's email record
F9	eml_file_next()	Move to the next client's email record
shift-F11	print_screen()	print the screen to a nominated printer
ESC, F12		return to the client maintenance screen

18.8 ESC_ESCALATION_DETAILS

coded in: mds0jesc.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, 'ESC' contact line selected, shift-F7 pushed
raison d' etre: set-up and maintain escalations for the client, escalations being hierarchies of messages sent one after another until receipt acknowledgement is received.

screen:



database tables:

mdsescal
mdsline
mdspager
mdstext
mdsextco

18.8.1 PROGRAM & SCREEN INITIALISATION

- first esc_escalation_details() calls esc_set_screen_array() which makes the 6 columns of editable text-boxes
- esc_escalation_details() constructs the remainder of the screen, the top "Line No:..." line and the bottom "F2 – READ ..." lines being represented as separate windows.
- finally control is passed to esc_process_escalation() which implements the escalation maintenance logic. Escalation maintenance is constrained (only) by the screen layout and the functionality bound to special keyboard keys.

18.8.2 ESCALATION MAINTENANCE: FUNCTION: ESC_PROCESS_ESCALATION

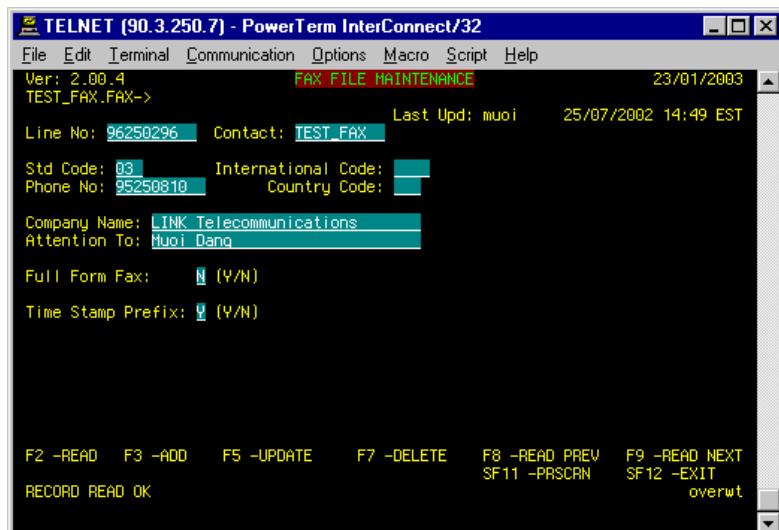
First esc_file_read() is called to read existing details from the escalation (mdsescal) database table. Then, in an infinite loop, key presses are read by edit_scrn(), the following keys being passed back (here) to be handled. Note that escalation maintenance is constrained (only) by the screen layout and the functionality bound to special keyboard keys

Key	Function Called	Description
shift-F1	esc_canned_text()	Print the actual 'canned-text' that will be sent as the escalation message.
F2	esc_file_read()	read the mdsescal database table (again).
F3	esc_file_add()	write a new escalation record to the mdsescal database table.
shift-F4	esc_del_line()	delete a single escalation line. F7 deletes all the escalation lines
F5	esc_file_update()	write changes back to the same escalation record.
shift-F5	esc_ins_line()	insert a line into the escalation
F6	esc_copy_escalation()	take a copy of an escalation record (for pasting as a new record)
shift-F6	esc_paste()	paste the previously taken copy as a new escalation record.
F7	esc_file_delete()	delete all the escalation records for this client
shift-F7		if the insertion-point is on the line number, return to the client maintenance screen.
F8	esc_file_prev()	move to the previous escalation in the database.
F9	esc_file_next()	move to the next escalation in the database.
Page Up	esc_prev_page()	scroll the list of escalations up
Page Down	esc_next_page()	scroll down the list of escalations.

18.9 FAX_FAX_DETAILS

coded in: mds0jfax.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, 'FAX' contact line selected, shift-F7 pushed
raison d' etre: set-up and maintain an fax contact for the client, where the client receives messages as faxes

screen:



database tables:
mdsfax

18.9.1 PROGRAM & SCREEN INITIALISATION

- fax_fax_details() constructs all of the static text and creates sub-windows for the top two and bottom two lines of the screen.
- fax_fax_details() then calls fax_process_fax().
- fax_process_fax() calls fax_set_screen_array() to prepare the writeable areas on the screen. The screen is now complete

18.9.2 FAX MAINTENANCE: FUNCTION: FAX_PROCESS_FAX

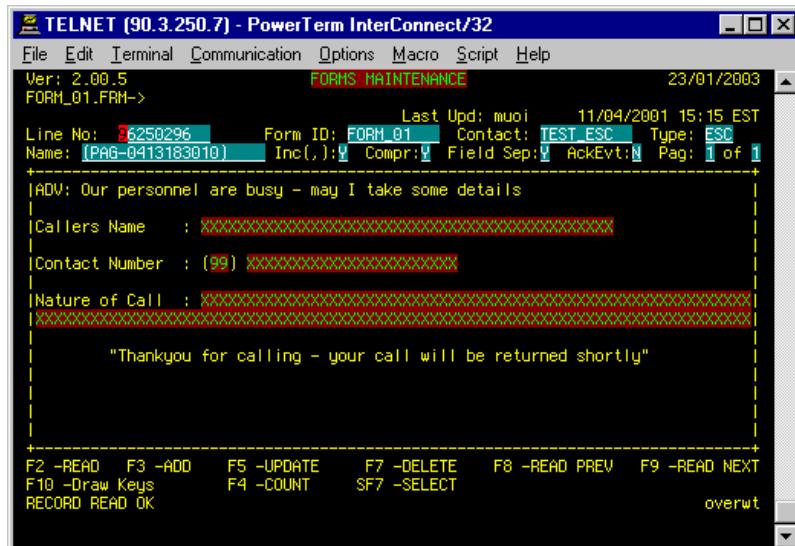
After fax_set_screen_array() completes construction of the screen, fax_file_read() is called to read existing fax details from the mdsfax database table. Next, in an infinite loop, key presses are read by edit_scrn(), the following keys being passed back to fax_process_screen to be handled. Note that fax maintenance is constrained (only) by the screen layout and the functionality bound to special keyboard keys

Key	Function Called	Description
F2	fax_file_read()	read the mdsfax database table (again).
F3	fax_file_add()	write a new fax record to the mdsfax database table.
F5	fax_file_update()	write changes back to the same fax record.
F7	fax_file_delete()	delete this fax record.
shift-F7		if the insertion-point is on the line number, return to the client maintenance screen.
F8	fax_file_prev()	move to the previous fax in the database.
F9	fax_file_next()	move to the next fax in the database.
shift-F11	print_screen()	print a copy of the screen to a nominated printer.

18.10 FRM_FORM_DETAILS

coded in: mds0jfrm.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, 'FRM' contact line selected, shift-F7 pushed
raison d' etre: set-up and maintain forms for the client, forms being customised interfaces made for some clients to facilitate unique data collection.

screen:



database tables:

mdsforms
mdsfmxtxt
mdsline
mdspager

18.10.1 PROGRAM & SCREEN INITIALISATION

- All of the static screen construction, quite a lot of it, is done in frm_form_details()
- frm_form_details() then calls frm_display_base_screen() to write the two "Line No:..." lines
- control is passed to frm_process_form() which uses frm_set_screen_array() to construct remainder of the screen, the other editable text-boxes, and frm_file_read() to read in the existing form (if any).

18.10.2 FORM MAINTENANCE: FUNCTION: FRM_PROCESS_FORM

After frm_process_form has finished constructing the window it starts an infinite loop, using edit_scrn() to read the user's key-presses, the special keys in the following table being passed back here to be handled. There are two sets of keystrokes here, the standard contact editing keys and form editing keys

Key	Function Called	Description
F2	frm_file_read()	read the mdsforms database table (again).
F3	frm_file_add()	write a new form record to the mdsforms database table.
F4	frm_count_enterable_char()	How many characters make up this form
F5	frm_file_update()	write changes back to the same form record.
F7	frm_file_delete()	delete all the form records for this client
shift-F7	if the insertion-point is on the line number, return to the client maintenance screen.	
F8	frm_file_prev()	move to the previous form in the database.

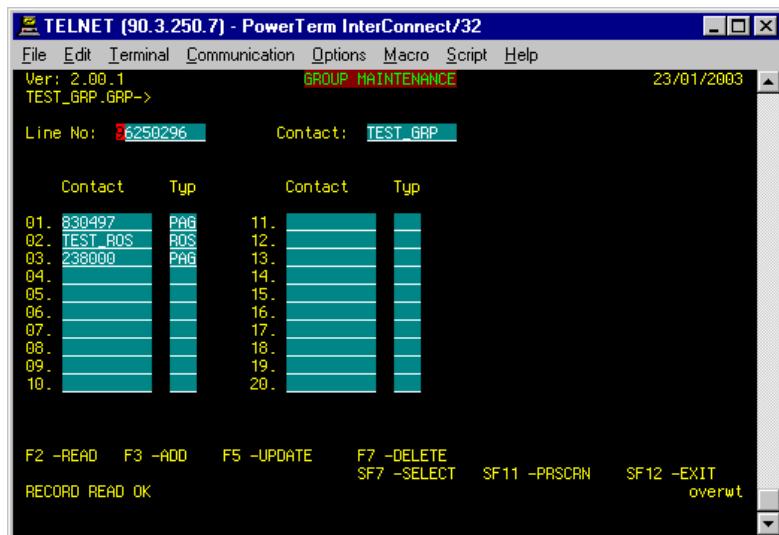
Form Editing Keys

	repeatedly insert a 'X' at the current position. Text will be entered here.
shift-F2	repeatedly insert a '9' at the current position. A number will be entered here.
shift-F3	repeatedly insert a '?' at the current position. A system reference number will be entered here.
shift-F4	frm_del_line() Delete a line in the form
shift-F5	frm_ins_line() insert a line into the form
F6	frm_copy() copy a line on the form; for later pasting
shift-F6	frm_paste() past a line (copied earlier with F6) onto the form.
page-down	frm_next_page() scroll down the form
page-up	frm_page_up() scroll up the form

18.11 GRP_GROUP_DETAILS

coded in: mds0jgrp.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, ‘GRP’ contact line selected, shift-F7 pushed
raison d’etre: set-up and maintain group paging for the client, groups being one message sent to many destinations

screen:



database tables:

mdsconta
mdspager

18.11.1 PROGRAM & SCREEN INITIALISATION

- grp_group_details() constructs most of the screen, the top “Line No:...” line and the bottom “F2 –READ ...” lines being represented as separate windows. It then calls grp_process_group() to both complete construction of the screen, and then process user interaction.
- grp_process_group() calls grp_set_screen_array() to generate all the 40 or so editable text boxes and grp_file_read() to gather and display the existing members of this group. grp_file_read() uses grp_get_contacts() to actually read the group members from the contact (mdsconta) database table.

18.11.2 GROUP MAINTENANCE: FUNCTION: GRP_PROCESS_GROUP

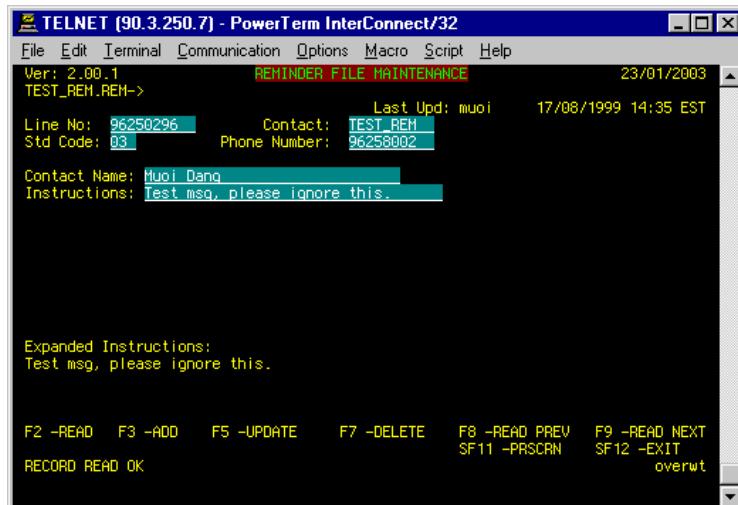
After the window has been constructed and any existing group members read, grp_process_group() manages the user interactions. As usual, key presses are read by edit_scrn(), some keys being passed back (here) to be handled. Note that group maintenance is constrained (only) by the screen layout and the functionality bound to special keyboard keys

Key	Function Called	Description
F2	grp_file_read()	re-read the group members
F3	grp_file_add()	add a new group member.
F5	grp_file_update()	write changes back over the existing group data
F7	grp_file_delete()	delete all the group records for this client
shift-F7	if the insertion-point is on the line number, return to the client maintenance screen.	
F8	grp_file_prev()	move to the previous group in the database.
F9	grp_file_next()	move to the next group in the database.
shift-F11	print_screen()	print a copy of the screen to a nominated printer.

18.12 REM_Reminder_Details

coded in: mds0jrem.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, 'REM' contact line selected, shift-F7 pushed
raison d' etre: set-up and maintain reminders, reminders being messages to the Call Centre Operators

screen:



database tables:
mdsphone

18.12.1 Program & Screen Initialisation

- remReminderDetails() constructs most of the screen, the top "Line No:..." line and the bottom "F2 -READ ..." lines being represented as separate windows. It then calls remProcessRem() to both complete construction of the screen, and then process user interaction.
- remProcessRem() calls remSetScreenArray() to generate the editable text-boxes and remFileRead() to read any existing reminder details from the phone (mdsphone) database table.

18.12.2 Reminder Maintenance: Function: REM_PROCESS_Rem

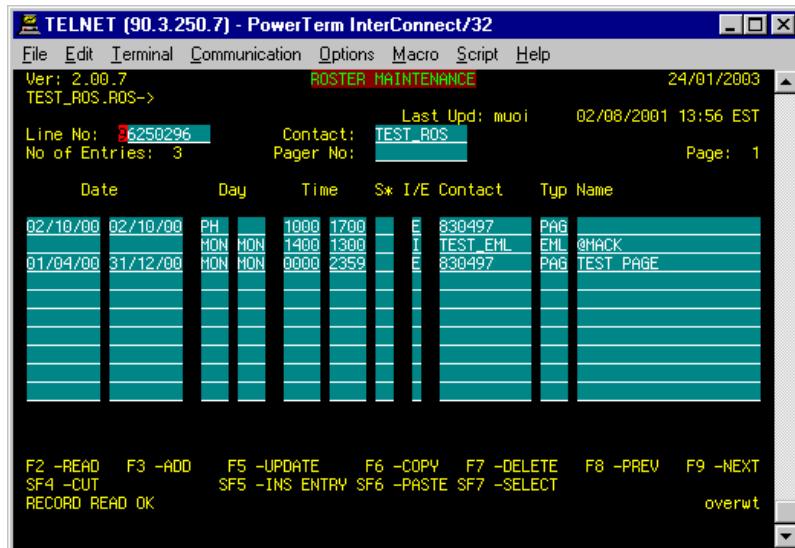
After remFileRead() is called to read existing details from the reminder (mdsphone) database table. Then, in an infinite loop, key presses are read by editScrn(), the following keys being passed back to be handled. Note that reminder maintenance is constrained (only) by the screen layout and the functionality bound to special keyboard keys

Key	Function Called	Description
F2	remFileRead()	re-read the mdsphone database table.
F3	remFileAdd()	write a new reminder record to the mdsphone database table.
F5	remFileUpdate()	write changes back to the same reminder record.
F7	remFileDelete()	delete the reminder record.
shift-F7	if the insertion-point is on the line number, return to the client maintenance screen.	
F8	remFilePrev()	move to the previous reminder in the database.
F9	remFileNext()	move to the next reminder in the database.
shift-F11	printScreen()	print a copy of the screen to a nominated printer.

18.13 ROS_ROSTER_DETAILS

coded in: mds0jros.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, ‘ROS’ contact line selected, shift-F7 pushed
raison d’ etre: set-up and maintain rosters for the client, rosters being time and date dependent message delivery behaviours

screen:



database tables:

mdsnrost
mdspager
mdsextco

18.13.1 PROGRAM & SCREEN INITIALISATION

- first ros_roster_details() calls ros_set_screen_array() which makes those columns of editable text-boxes.
- next ros_roster_details() constructs the remainder of the screen, the top “Line No:...” line and the bottom “F2 – READ ...” lines being represented as separate windows.
- finally control is passed to ros_process_roster() to implement the roster maintenance logic.

18.13.2 ROSTER MAINTENANCE: FUNCTION: ROS_PROCESS_ROSTER

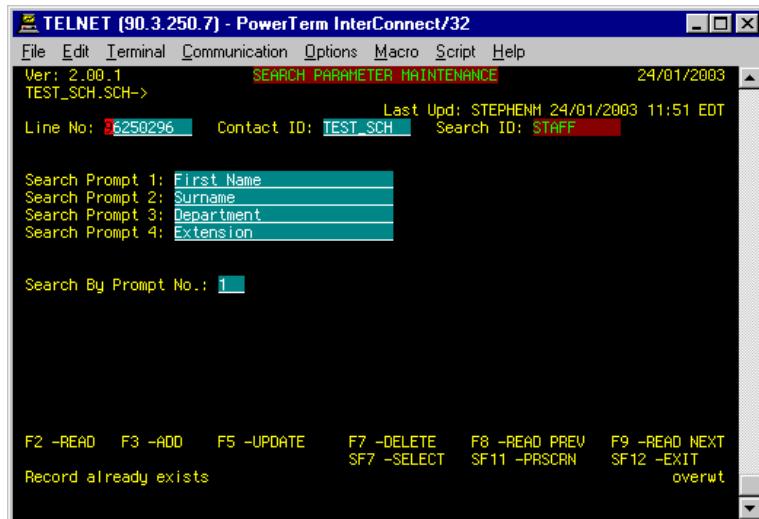
First ros_file_read() is called to read the clients existing roster information from the roster (mdsnrost) database table. Then, in an infinite loop, key presses are read by edit_scren(), the following keys being passed back (here) to be handled. Note that roster maintenance is constrained (only) by the structure of the screen layout and the functionality bound to special keyboard keys

Key	Function Called	Description
F2	ros_file_read()	read the mdsnrost database table (again).
F3	ros_file_add()	write a new roster record to the mdsnrost database table.
shift-F4	ros_del_line()	delete a single roster line. F7 deletes all the roster lines
F5	ros_file_update()	write changes back to the same roster record.
shift-F5	ros_ins_line()	insert a line into the roster
F6	ros_copy_roster()	take a copy of an roster record (for pasting as a new record)
shift-F6	ros_paste()	paste the previously taken copy as a new roster record.
F7	ros_file_delete()	delete all the roster records for this client
shift-F7	if the insertion-point is on the line number, return to the client maintenance screen.	
F8	ros_file_prev()	move to the previous roster in the database.
F9	ros_file_next()	move to the next roster in the database.
Page Up	ros_prev_page()	scroll the list of rosters up
Page Down	ros_next_page()	scroll down the list of rosters.

18.14 SCH_SEARCH_PAR_DETAILS

coded in: mds0jschp.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, ‘SCH’ contact line selected, shift-F7 pushed
raison d’etre: set-up and maintain search question relevant to the indial. Search questions are asked by the Call Centre Operator to the caller to find the correct message destination.

screen:



database tables:

mdsextco
mdsschd
mdspager

18.14.1 PROGRAM & SCREEN INITIALISATION

- schd_search_par_details() constructs all of the static text on the form before calling sch_process_schp() to handle the user’s interaction.
- first sch_process_schp() calls schp_set_screen_array() to construct the four editable text boxes.
- next it uses schp_file_read() to initialise these boxes with existing search data from the database.

The screen is now complete. The remainder of sch_process_schp() manages the input and implements the effects of the user’s keystrokes.

18.14.2 SEARCH DATA MAINTENANCE: FUNCTION: SCHP_PROCESS_SCHP

sch_process_schp(), in an infinite loop, reads the user’s key press (with edit_scrn()), the following keys being returned back to be handled.

Key	Function Called	Description
F2	schp_file_read()	re-read the mdsextco and mdsschd database tables.
F3	schp_file_add()	write a new search record to the mdsextco and mdsschd database tables.
F5	schp_file_update()	write changes back to the same search record.
F7	schp_file_delete()	delete all the search records for this client
shift-F7	if the insertion-point is on the line number, return to the client maintenance screen. if the insertion-point is on a “search-prompt” field, call sch_search_dat_details() (q.v.) (in mds0jschd.c) to add or edit the search data.	
F8	schp_file_prev()	move to the previous search in the database.
F9	schp_file_next()	move to the next search in the database.
shift-F11	print_screen()	print the screen to a nominated printer

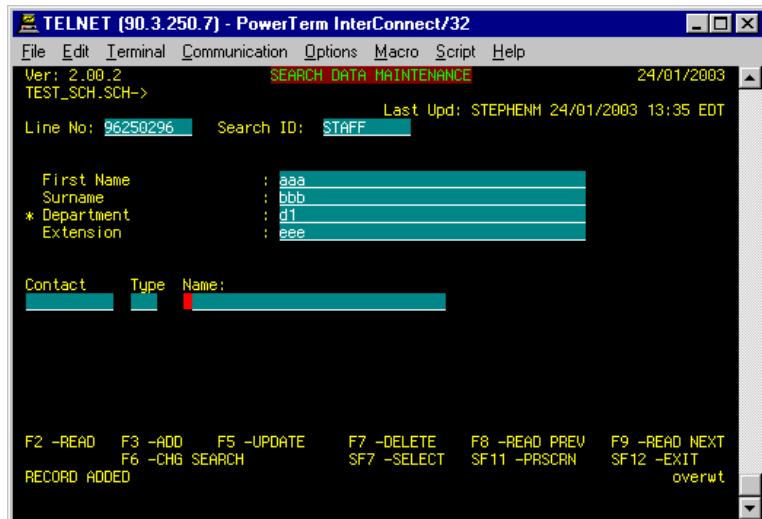
18.15 SCH_SEARCH_DAT_DETAILS

coded in: mds0jschd.c

called from (code): schp_process_schp in file mds0jschp.c

called from (window): Search Parameter Maintenance, cursor on a search prompt, push shift-F7
raison d' etre: maintain the possible matches for a message-destination search

screen:



database tables:

mdsextco

mdsschd

mdspager

18.15.1 PROGRAM & SCREEN INITIALISATION

- schd_search_dat_details() constructs all of the static text on the form before calling sch_process_schd() to handle the user's interaction.
- first sch_process_schd() calls schd_set_screen_array() to construct the nine editable text boxes.
- next it uses schd_file_read() to initialise these boxes with existing search data from the database.

The screen is now complete. The remainder of sch_process_schd() manages the input and implements the effects of the user's keystrokes.

18.15.2 SEARCH DATA MAINTENANCE: FUNCTION: SCHD_PROCESS_SCHD

The remainder of schd_process_schd() runs an infinite loop. In each iteration it first calls schd_display_record() to display any records available that match what the user has typed so far. Next it uses edit_scren() to read the user's key presses, the following keys being returned back here to schd_process_schd() to be handled.

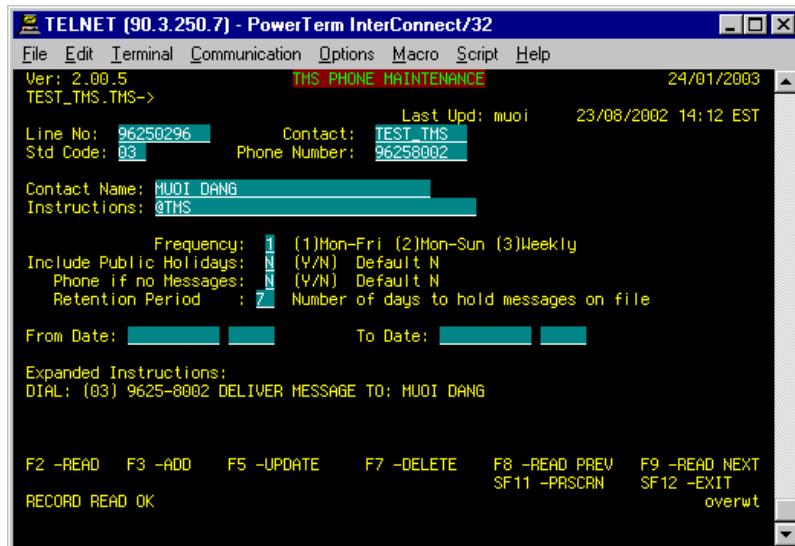
Key	Function Called	Description
F2	schd_file_read()	re-read the mdsextco and mdsschd database tables.
F3	schd_file_add()	write a new search record to the mdsextco and mdsschd database tables.
F5	schd_file_update()	write changes back to the same search record.
F6	schd_chg_search()	find if there are message destinations matching the fields entered, ...
	schd_file_read()	... and, if so, read them from the database
F7	schd_file_delete()	delete all the search records for this client
shift-F7	if the insertion-point is on the line number, return to the client maintenance screen. if the insertion-point is on a "search-prompt" field, call sch_search_dat_details() (in mds0jschd.c) to add or edit the search data.	
F8	schd_file_prev()	move to the previous search in the database.
F9	schd_file_next()	move to the next search in the database.
shift-F11	print_scren()	print the screen to a nominated printer

Upon completion, schd_process_schd() returns, using this dat to populate the search parameter screen.

18.16 TMS_TMS_DETAILS

coded in: mds0jtms.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, 'TMS' contact line selected, shift-F7 pushed
raison d' etre: set-up and maintain text message services for the client, text message services being a standard phone answering / message taking service.

screen:



database tables:

mdsphone
mdstimev
mdsline

18.16.1 PROGRAM & SCREEN INITIALISATION

- tms_tms_details() constructs all of the static text on the form before calling tms_process_tms() to handle the user's interaction.
- tms_process_tms() first calls tms_set_screen_array() to construct the editable text boxes.
- next it uses tms_file_read() to initialise these boxes with existing tms data from the database.

18.16.2 TEXT MESSAGE MAINTENANCE: FUNCTION: TMS_PROCESS_TMS

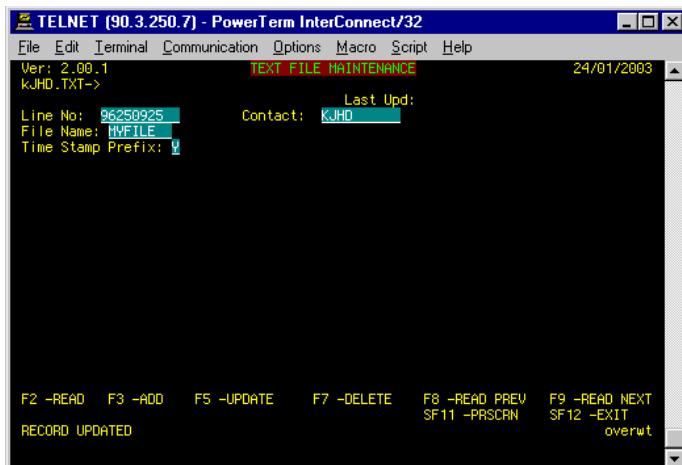
After tms_set_screen_array() and tms_file_read() have been finished construction of the screen, tms_process_tms(), in an infinite loop, calls edit_sscrn() to read key presses, the following keys being passed back to be handled.

Key	Function Called	Description
F2	tms_file_read()	re-read the mdsphone database table.
F3	tms_file_add()	write a new text message service record to the mdsphone database table.
F5	tms_file_update()	write changes back to the same text message service record.
F7	tms_file_delete()	delete all the text message service records for this client
shift-F7	if the insertion-point is on the line number, return to the client maintenance screen.	
F8	tms_file_prev()	move to the previous text message service in the database.
F9	tms_file_next()	move to the next text message service in the database.
shift-F11	print_screen()	print the screen to a nominated printer

18.17 TEXT_TXT_DETAILS

coded in: mds0jtxt.c
called from (code): select_contact() in skymds00j.c
called from (window): client maintenance window, 'TXT' contact line selected, shift-F7 pushed
raison d' etre: set-up and maintain text message services. Like TMS, but with the messages stored in a file and optionally emailed to the client

screen:



database tables:

mdstext
mdsline

18.17.1 PROGRAM & SCREEN INITIALISATION

- txt_text_details() constructs most of the screen, writing all of the static text.
- then txt_text_details() calls txt_process_txt() to finish construction of the screen and collect user input.
- txt_process_txt() calls txt_set_screen_array() to generate the writeable text boxes
- it then uses txt_file_read() to read in any existing 'Text' text message service data for the client

txt_process_txt() now processes the user's input.

18.17.2 'TEXT' MESSAGE MAINTENANCE: FUNCTION: TXT_PROCESS_TEXT

Like the other contact management screens, txt_process_text() uses edit_scren() to retrieve the keystrokes from the user, passing most of them back here, the ones in the following table editing the text message service records for the client.

Key	Function Called	Description
F2	txt_file_read()	read the mdstext database table again.
F3	txt_file_add()	write a new text message service record to the mdsDDBB database table.
F5	txt_file_update()	write changes back to the same text message service record.
F7	txt_file_delete()	delete all the text message service records for this client
shift-F7	if the insertion-point is on the line number, return to the client maintenance screen.	
F8	txt_file_prev()	move to the previous text message service in the database.
F9	txt_file_next()	move to the next text message service in the database.
shift-F11	print_screen()	print the screen to a nominated printer

19 File Menu programs

The “file” menu “is” a call to file_menu(), which presents the menu (and manages spawning the process) which concern the overall set-up of the <System_Name> system. The menu items and programs are hard coded (as they will seldom change) though which menu-items end up appearing for each user is individually determined based on the user’s security settings.



Menu Item	Program	Comment
User/Password Maintenance	skymds00b	edit (add, delete, change) a password.
Operator Bulletin Maintenance	skymds00d	Edit the bulletin that appears on every user log-in.
Public Holiday Maintenance	skymds00p	The public holiday screen set-up / maintenance screen.
Function Code Maintenance	skymds39	assign access to parts of <System_Name> to groups of users.
Security Group Maintenance	skymds32	assign members and access attributes to groups
Audit Trail	skymds77	Display edits to ‘contact’ ³⁶ related database tables
Software Request Form Maintenance	skymds82	No longer relevant. Do not use.
Reminder Code Maintenance	skymds34	Set up details of pre-set reminder patterns
Email Blocking Maintenance	mdsemblk	Stop users or domains using <System_Name> email services
Destination File Maintenance	mdsdestm	(Queues) Maintain details of <System_Name> message queues.

Table 109: skymds00: File Menu actions

Now, in an infinite loop, the user’s choice(s) is/are processed.

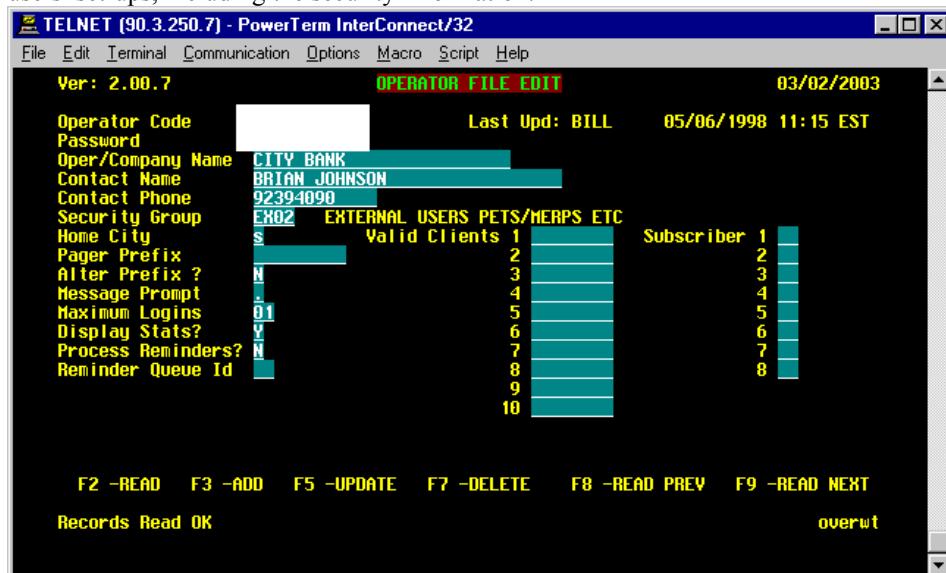
- rev_select() reads the user’s menu selection
- exec_prog() is used to run the appropriate subprogram. See the table above.
- PORT_update_port_rec() is used update the details in the mdsports database table to indicate that the current running process is no-longer whatever was spawned, but (back to) skymds00 again.
- Once the user exits the memory allocated for the menu items is freed.

³⁶ Advanced processing

19.1 “USERNAME / PASSWORD MAINTENANCE”: SKYMD00B

coded in: skymds00b.c
called from (code): file_menu() in skymds00.c
called from (window): Base <System_Name> menu, file sub-menu selected, “Username / password Maintenance” selected from sub-menu.
raison d’ etre: This is actually a full “Operator Maintenance” application, giving access to many details of users’ set-ups, including the security information.

screen:



database tables:

mdspassw
mdspager
mdsrem
mdsports

19.1.1 PROGRAM & SCREEN INITIALISATION

- main() call initialise(), and then process_screen() to display the screen and manage user interactions
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin WINDOW structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
- process_screen() sets a few defaults and calls display_screen(). display_screen() generates the “Username / password Maintenance” curses screen (see the screen-shot above).
- display_screen() checks the security of the user, before using the display_scn() utility to display the screen.

The screen now constructed, control returns to process_screen() to implement the “Username / password Maintenance” functionality

19.1.2 FUNCTION: PROCESS_SCREEN

After setting some defaults and calling display_screen(), process_screen(): uses edit_scn() to retrieve the user’s key

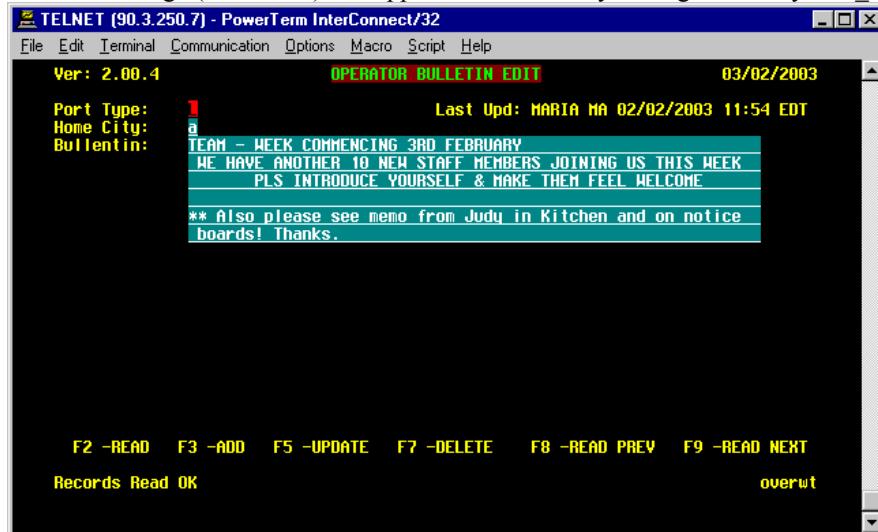
- uses edit_scn() to retrieve which key the user pushed
- calls the appropriate function, as per the following table

Key	Function Called	Description
F2	read_operator()	collect and display the details for the “Operator Code” operator
F3	add_operator()	check and add a new <System_Name> operator. Update to the maintenance queue.
F5	update_operator()	save changes to the current operator and update to the maintenance queue.
F7	delete_operator()	delete the current operator
F8	prev_operator()	move to the previous operator in the database.
F9	next_operator()	move to the next operator in the database.

19.2 “OPERATOR BULLETIN MAINTENANCE”: SKYMDSD0D

coded in: skymds00d.c
called from (code): file_menu() in skymds00.c
called from (window): Base <System_Name> menu, file sub-menu selected, “Operator Bulletin Maintenance” selected.
raison d’ etre: Edit the message (“Bulletin”) that appears each time anyone logs onto <System_Name>

screen:



database tables:

mdsports
mdsbull

19.2.1 PROGRAM & SCREEN INITIALISATION

- main() call initialise(), and then process_screen() to display the screen and manage user interactions
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin WINDOW structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
- process_screen() calls display_screen() which displays the “Bulletin Maintenance” curses screen.

19.2.2 FUNCTION: PROCESS_SCREEN

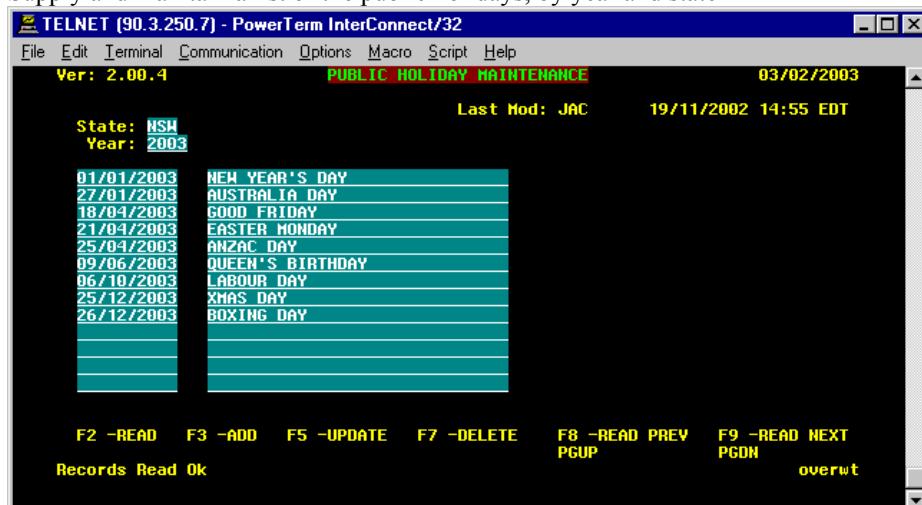
In an infinite loop, process_screen:

- calls display_screen()to update the screen,
- edit_scrn to obtain any function keys pushed
- process the function keys as per the following table:

Key	Function Called	Description
F2	read_bulletin()	read the next bulletin from the mdsbull table
F3	add_bulletin()	check (check_bulletin())and add a bulletin.
F5	update_bulletin()	save changes to the current bulletin and update to the maintenance queue.
F7	delete_bulletin()	delete the current bulletin
F8	prev_bulletin()	move to the previous bulletin in the database.
F9	next_bulletin()	move to the next bulletin in the database.

19.3 “PUBLIC HOLIDAY MAITENANCE”: SKYMD00P

coded in: skymds00p.c
called from (code): file_menu() in skymds00.c
called from (window): Base <System_Name> menu, file sub-menu selected, “Public Holiday Maintenance” selected.
raison d’ etre: Supply and maintain a list of the public holidays, by year and state
screen:



database tables:

mdspubh
mdsports

19.3.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and process_screen() to display the screen and manage user interactions
- initialise():
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin WINDOW structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
- process_screen() sets a few defaults before calling display_screen() to generate the “Public Holiday Maintenance” curses screen (see the screen-shot above).
- display_screen(), basically one line of code, uses display_scrn() to actually show the screen.

19.3.2 FUNCTION: PROCESS_SCREEN

After clearing out a few fields and setting a few defaults process_screen(), in a loop:

- uses display_screen() to (re-) display any public holiday updates
- calls edit_scrn() to manage the user’s editing, and passes back the function key presses, where they are handled:

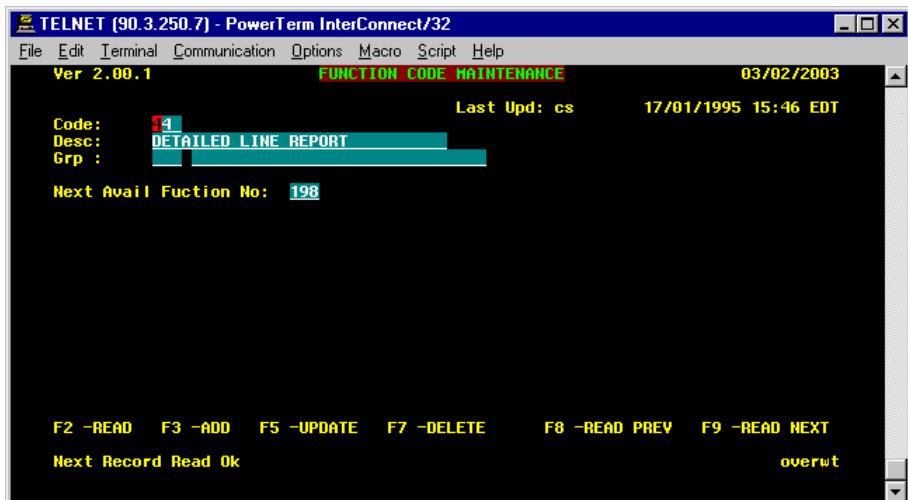
Key	Function Called	Description
F2	read_pubh()	use load_holiday_array() to read the public holiday records from mdspubh
F3	add()	call add_dbase_recs() to write the public holiday records to mdspubh.
F5	update()	validate_records(), then store_records() to save public holiday date changes.
F7	delete()	use delete_dbase_recs() to delete all the records for the selected state and year.
F8	read_prev()	read the previous public holiday data (the previous year).
F9	read_next()	move to the next bulletin in the database.
Page-Down	next_page()	reads, and displays, the next page of public holiday dates
Page-Up	prev_page()	reads, and displays, the previous page of public holiday dates

Note that a “holiday_array” structure is used as an interim public holiday holding structure between the display and the database. This has probably been done to minimise the number of database accesses.

19.4 “FUNCTION CODE MAINTENANCE”: SKYMD39

coded in: skymds39.c
called from (code): file_menu() in skymds00.c
called from (window): Base <System_Name> menu, file sub-menu selected, “Function Code Maintenance” selected.
raison d’etre: Maintain a list of numbers (“function codes”) representing parts (200-ish) of the <System_Name> system, used in approving security access to these parts of <System_Name>.

screen:



database tables:

mdsports
mdspassw
mdsfunct

19.4.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
- process_screen() calls display_screen() to generate the “Function Code Maintenance” curses screen.
- display_screen(), with a few bits of house keeping, just uses wrefresh() to actually show the window.

The screen now constructed, control returns to process_screen() to implement the “Function Code Maintenance” functionality

19.4.2 FUNCTION: PROCESS_SCREEN

In a loop, process_screen():

- calls display_screen() to show the new / updated function code record
- uses edit_scren() to retrieve the user’s key press. If it is a function key, call a function as per the table:

Key	Function Called	Description
F2	read_funct()	read the next ‘Function Code’ record from the mdspunct table
F3	add_funct()	add a ‘Function Code’ record, and copy the update to the maintenance queue.
F7	delete_funct()	With confirmation, delete the current ‘Function Code’ record
F8	prev_funct()	move to the previous ‘Function Code’ record in the database.
F9	next_funct()	move to the next ‘Function Code’ record in the database.

19.5 “SECURITY GROUP MAINTENANCE”: SKYMD32

coded in: skymds32
called from (code): file_menu() in skymds00.c
called from (window): Base <System_Name> menu, file sub-menu selected, “Security Group Maintenance” selected.

raison d’etre: Give defined groups access to parts of the <System_Name> system, via the “Function Codes” defined on the “Function Code Maintenance” window.

screen:



database tables:

mdsfunct
mdsserec
mdspassw

19.5.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
 - read in the function codes (mdsfunct database table) into memory
- process_screen() sets a few defaults and calls display_screen() to finish constructing, and then display, the “Security Group Maintenance” window.

19.5.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

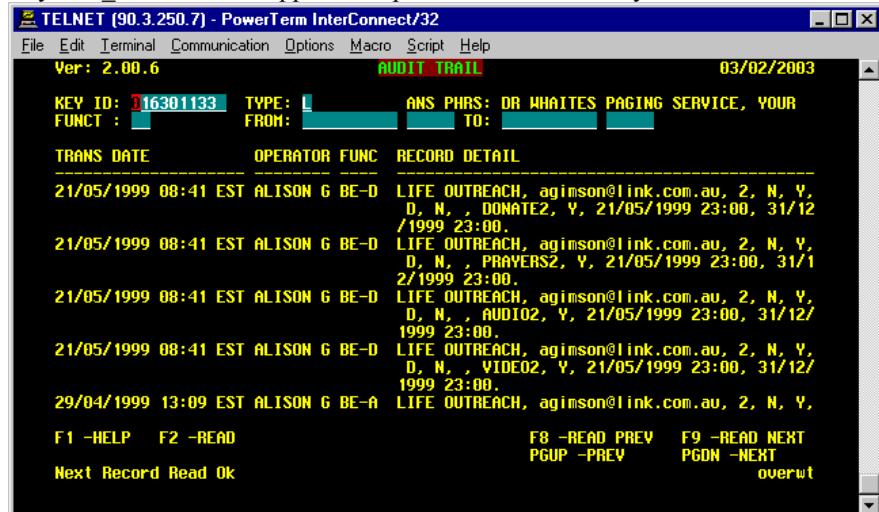
- calls display_screen() to re-display the “Security Group Maintenance” data, possibly updated
- uses edit_scrn() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_secur()	collect and display the Security Maintenance Codes
F3	add_secur()	add a new security group, with a ‘1’ or ‘0’ assigned to every function code.
F5	update_secur()	save changes to the current security group and update to the maintenance queue.
F7	delete_secur()	delete the current Security Group
F8	prev_secur()	move to the previous Security Group in the database.
F9	next_secur()	move to the next Security Group in the database.
Page-Down	next_page()	reads, and displays, the next Security Maintenance page
Page-Up	prev_page()	reads, and displays, the previous Security Maintenance page

19.6 “AUDIT TRAIL”: SKYMDS77

coded in: skymds77.c
called from (code): file_menu() in skymds00.c
called from (window): Base <System_Name> menu, file sub-menu selected, “Audit Trail” selected from sub-menu.
raison d’ etre: Records (“audit trails”) are kept of every change to message-setup information in <System_Name>. This application provides a user-friendly interface to the audit trail.

screen:



database tables:

mdsaudit
mdspager
mdsline
mdspassw
mdsports

19.6.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
- process_screen() calls display_screen() to finish constructing, and then display, the “Audit Trail” screen.

The screen now constructed, control returns to process_screen() to implement the “Audit Trail” functionality.

19.6.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

- calls display_screen() to re-display the “Security Group Maintenance” data, possibly updated
- uses edit_scrn() to retrieve the user’s key press. If it is a function key, a function is called as per the table:

Key	Function Called	Description
F2	read_audit()	read the next Audit Trail record from the mdsbull table
F8	prev_audit()	move to the previous Audit Trail record in the database.
F9	next_audit()	move to the next Audit Trail record in the database.

“REMINDER PARAMETER MAINTENANCE”: SKYMD34

coded in: skymds34.c
called from (code): file_menu() in skymds00.c
called from (window): Base <System_Name> menu, file sub-menu, “Reminder Parameter Maintenance” selected.
raison d’ etre: A reminder is an instruction to a Call Centre Operator to perform some task, the reminders being queued (like messages) and each Call Centre (‘node’) having 6 reminder queues (‘queue’ = ‘index’). Sometimes reminders need to be diverted to different nodes or queues. This screen manages reminder diversion.

screen:



database tables:

mdsremp
mdspassw
mdsports

19.6.3 PROGRAM & SCREEN INITIALISATION

- main() calls initialise(), and then process_screen() to display the screen and manage user interactions.
- initialise():
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
 - attach to the Public Holiday Shared memory
- process_screen() calls display_screen() to finish constructing, and then display, the “Reminder Parameter Maintenance” window.
- display_screen() loads any current data onto the display and calls wrefresh to (re-) display it.

19.6.4 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

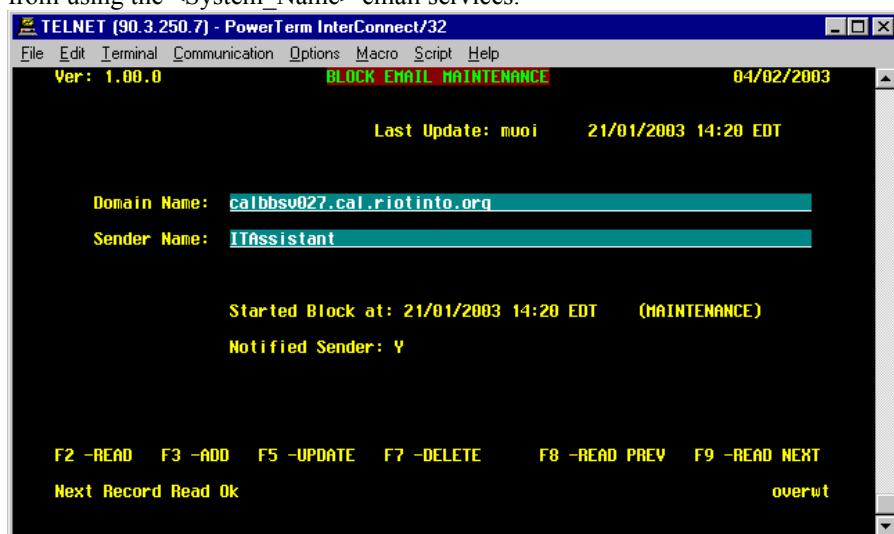
- calls display_screen() to re-display the Reminder Parameter data, possibly updated
- uses edit_scren() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_remp()	read the next reminder code from the mdsremp table. Uses validate_data().
F3	add_remp()	check (validate_data()) and add a reminder code.
F5	update_remp()	save changes to the current reminder code and update to the maintenance queue.
F7	delete_remp()	delete the current reminder code
F8	prev_remp()	move to the previous reminder code in the database.
F9	next_remp()	move to the next reminder code in the database.
also:	validate_data()	checks the entered diversion dates, queue and node.

19.7 “EMAIL BLOCKING MAINTENANCE”: MDSEMBLK

coded in: mdseblk.c
called from (code): file_menu() in skymds00.c
called from (window): Base <System_Name> menu, file sub-menu selected, “Email Blocking Maintenance” selected from sub-menu.
raison d’ etre: Some email addresses, or entire domains, have to be temporarily or permanently prohibited from using the <System_Name> email services.

screen:



database tables:

mdsports
mdseblk

19.7.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then select_report() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
- select_report() calls display_screen() to finish constructing and display the “Email Blocking Maintenance” window. display_screen() does some final formatting before using wrefresh() to show the window.

19.7.2 FUNCTION: SELECT_REPORT

In an infinite loop, process_screen():

- calls display_screen() to re-display the Email Blocking data, possibly updated.
- uses edit_scren() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_block_email()	read the next Email Blocking record from the mdseblk table
F3	add_upd_block_email(1)	add a Email Blocking record.
F5	add_upd_block_email(0)	save changes to the current Email Blocking record.
F7	delete_block_email()	delete the current Email Blocking record
F8	prev_block_email()	move to the previous Email Blocking record in the database.
F9	next_block_email()	move to the next Email Blocking record in the database.

A Validate_screen() function exists, is called, but does nothing (returns true). Helpful for future development.

19.8 “DESTINATION FILE MAINTENANCE”: MDSDESCR

coded in: mdsdescr.c
called from (code): file_menu() in skymds00.c
called from (window): Base <System_Name> menu, file sub-menu, “Destination File Maintenance” selected.
raison d’ etre: Present and maintain the extensive information on the 100 or so <System_Name> message queues.

screen:



database tables:

mdsports
mdsdestm

19.8.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then run_maint_scr() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
- run_maint_scr() calls display_screen().
- display_screen() finishes constructing, and then displays, the “Destination File Maintenance” window.

The screen now constructed, control returns to run_maint_scr() to implement the “Destination File Maintenance” functionality

19.8.2 FUNCTION: RUN_MAINT_SCR

In an infinite loop, run_maint_scr():

- calls display_screen() to re-display the Destination Queue record data, possibly updated
- uses edit_scren() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_destm_rec()	read the next Destination Queue record from the mdsdestm table
F3	add_upd_destm_rec(1)	check (validate_data())and add a Destination Queue record.
F5	add_upd_destm_rec(0)	save changes to the current Destination Queue record.
F7	delete_destm_rec()	delete the current Destination Queue record
F8	prev_destm_rec()	move to the previous Destination Queue record in the database.
F9	next_destm_rec()	move to the next Destination Queue record in the database.

20 Report Menu

It's the same general procedure for the report menu as for the file menu. The report menu is instantiated by the `report_menu()` function, `report_menu()` waits for a user's selection (using `rev_select()`) and:

- either calls either the appropriate program,
- or undertakes one of the two printer management activities directly itself.



And behind all that gloss:

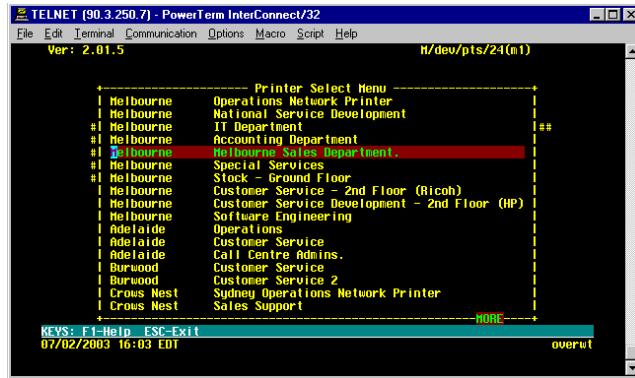
Menu Item	Program	Description
Set Printer	---	printer_select_menu(), which presents a list of <System_Name> printers.
Initialise Printer	---	Initialise the selected printer.
Report Generator	skymds46	Print out a report of data from the <System_Name> history. May also use one of: <ul style="list-style-type: none">□ skymds46a – report based on event or system reference number□ skymds46b – report based on contact ID□ skymds46c – report based on pager number□ skymds46b – report based on indial (phone) number
Pager Stats Report	skymds40	report on the number of calls to some pagers or account numbers.
Detailed Line Report	skymds47	'line' indial reporting. Uses skymds47a to access the database.
Answer Phrase Report	skymds48	all (yes, all) the answer phrases used, skymds48a accessing the database.
Batch Fax Setup	skymds19	Prepare batch faxes
Batch Fax List	skymds46e	Display a list of pending batch faxes.
Batch Email Setup	skymds19a	Prepare a batch email account.

20.1 “SET PRINTER”

If the first letter of the menu title is ‘S’ (yes, that’s how its done), the printer_select_menu() function is called:

printer_select_menu():

- reads in a pre-made list of printers and display the list in a window



- rev_select() retrieves the user’s choice
- the printer is actually set by copying the printer name into the lp_dest variable.
- the printer choice is copied to the maintenance queue

20.2 “INITIALISE PRINTER”

If the first letter of the command is ‘I’, the following takes place:

- a system() call runs the command: disable the_lp_printer >&2
- sleep for 2 seconds
- a system() call runs the command: enable the_lp_printer >&2

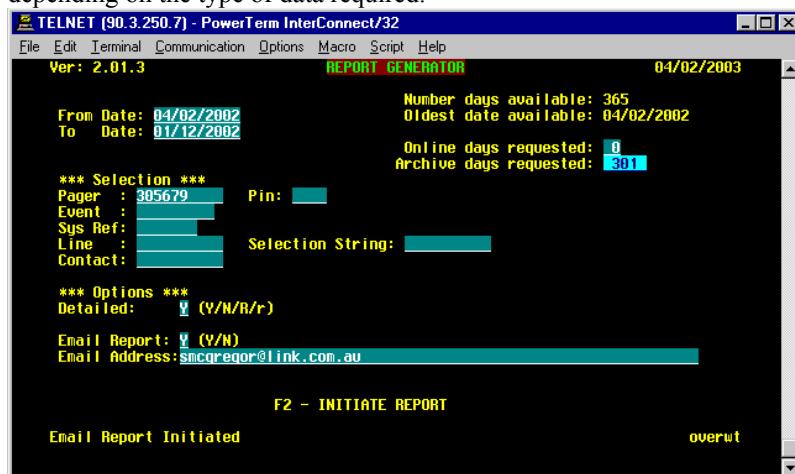
The the_lp_printer is the lp_dest variable set by choosing the printer during “Set Printer”

20.3 REPORT GENERATOR (HISTORY DATA): SKYMDS46

coded in: skymds46.c
called from (code): report_menu() in skymds00.c
called from (window): Base <System_Name> menu, report sub-menu selected, “Report Generator” selected from sub-menu.

raison d’ etre: A standard interface to retrieving history data. skymds46 calls one of four sub-programs depending on the type of data required.

screen:



20.3.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_report() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - opens its database tables
- process_report() calls display_scrn() to finish constructing, and then display, the “Report Generator” window.

20.3.2 FUNCTION: PROCESS_REPORT

process_report():

- calls display_scrn() and then waits for a F2 key-press.
- does a couple of date calculations
- calls spawn_report() which calls the appropriate sub program to report on the contact, pager, dates or whatever.

20.3.3 FUNCTION: SPAWN_REPORT

Spawn_report generates and optionally e-mails or prints a report as detailed on the “Report Generator” screen.

- call validate_screen() to ensure that the data supplied by the user makes sense. Basically either a pager number or a line number (indial) must be entered, plus a few other rules.
- either set up the printer, or prepare a temporary file if the data is to be emailed.
- call the first sub-program satisfying the “User entered a:” criteria in the following table:

User entered a:	program	Report Type	Between dates, report on:
pager number	skymds46c	“Pager Message Report”	All pager messages
Event Number	skymds46a	“Event Message Report”	All acknowledged escalations
System Ref. Number	skymds46a	“System Reference Number Report”	All (“system”) reference numbers quoted
Contact ID (indial)	skymds46b skymds46d	“Contact Message Report” “Line Message Report”	All contacts of specified type All messages to the specified indial

- Finally, if the “Email Report” field was set to ‘Y’, the report is emailed to the supplied address.

20.4 REPORT GENERATOR: SUB PROGRAMS

20.4.1 SKYMD546A: EVENT & SYSTEM REFERENCE NUMBER REPORTS

- initialise():
 - reads in environment variables
 - the command-line is parsed, which includes an event number or a system reference number
 - opens database tables.
- either get_header_rec_event() [events] or get_header_rec_sys_ref() [Sys. Ref. Number] is called to collect the information required
- print_headings() makes the report header
- Either process_event() [events] or process_sys_ref() [Sys. Ref. Number] is used to generate the appropriate report

These two functions read through the history data (using HIST_read_* functions), and call either print_step() or print_message() to write the history record to a report. print_step() details information about the message (date, contact details, operator – primary history type information) while print_message() writes out the actual message.

20.4.2 SKYMD546B: CONTACT MESSAGE REPORT

Detailed report on the “contacts” (specialised messaging services) used by the pager or indial.

- initialise():
 - reads in environment variables
 - parses the command-line,
 - opens database tables and connects to shared memory.
- process_contact():
 - calls print_headings() to, well, print the report headings
 - HIST_read_prim() finds the first primary history record for a contact of the specified type for this pager number in the specified time span. init_date_time() prepares the dates and times.
 - In a nested loop, each primary history record is read and (sub-loop) each of its associated secondary records is written out to the report. Note that it is only the secondary records that are written – they have all the interesting stuff anyway. print_step() (contact, message info) and print_message(the message) are used.

20.4.3 SKYMD546C: PAGER MESSAGE REPORT

skymds46c (Pager Message Report) prints even more detailed information about each pager message (and contact) belonging to the indial or pager during the stipulated period. Structurally it is similar to the other skymds46* programs,

- initialise():
 - reads in environment variables
 - parses the command-line, quite a number of options
 - opens database tables and connects to (security) shared memory
- process_pager() generates the pager call report.
 - call validate_pager() to check that the pager number and any supplied PIN is valid
 - locate the first appropriate primary history record, using NHIST_read_prim() instead of HIST_read_prim()
 - As before, in a nested loop, each primary history record is read, some of this data being used in the ‘print_step()’ function to print a heading for each primary and associated secondary records, following which each associated secondary history record is written. print_contact() and print_message() are used to print (secondary history) data referring to contact details or the message text respectively

20.4.4 SKYMD546D: LINE MESSAGE REPORT

The Line Message Report lists all the message sent to an indial over a set period. Similar to the other skymds46* progs.

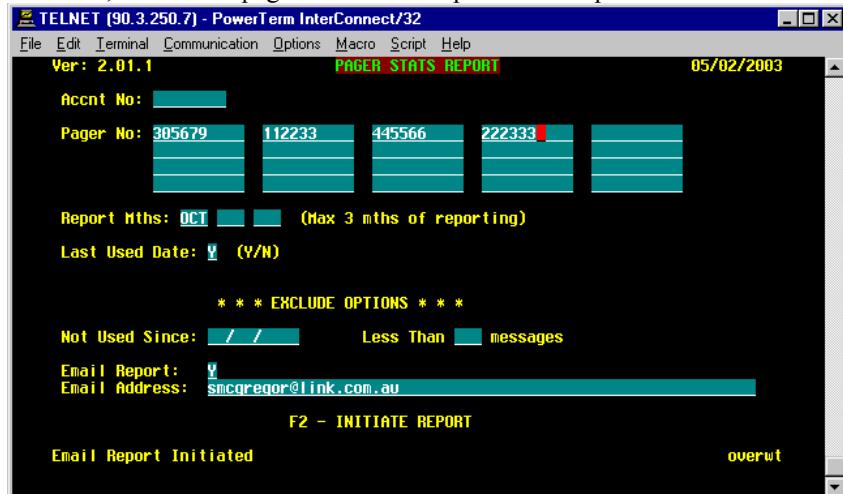
- initialise():
 - reads in environment variables
 - parses the command-line,
 - opens database tables and connects to shared memory.
- process_contact(), like in the other skymds46* programs:
 - reads, and writes out, every secondary history record (for each primary history record) over the period.
 - uses a print_step() to write the contact info, and print_message() to write the message.

20.5 PAGER STATS REPORT: SKYMDSD40

coded in: skymds40.c
called from (code): report_menu() in skymds00.c
called from (window): Base <System_Name> menu, report sub-menu, “Pager Stats Report” selected from sub-menu.

raison d’ etre: To read from the pager stats table (mdspasts) and produce report of pager usage for either the client, or for each pager listed. The report is either printed or emailed.

screen:



20.5.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_report() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the database tables
- process_report() sets a couple of defaults and calls display_screen() to finish constructing, and then display, the “Pager Stats Report” window.
- display_screen() actually constructs, and displays, the screen.

20.5.2 FUNCTIONS: PROCESS_REPORT, GENERATE_REPORT

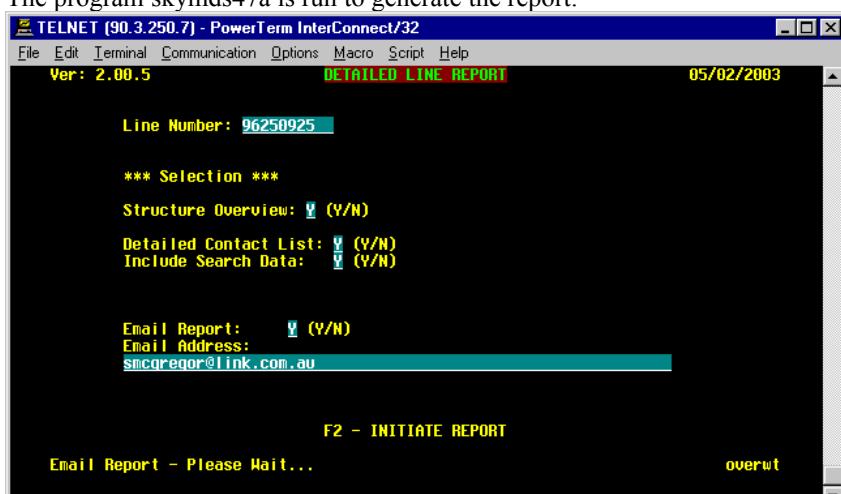
process_report() waits for an F2 key-press, and when it gets one it calls generate_report().

- generate_report():
 - uses validate_screen() to ensure that the screen has been properly filled out.
 - if a client number has been entered, calls process_client(), otherwise calls process_pager().
 - Once the report has been created generate_report() uses sendmail to email it, if email is requested.

20.5.3 FUNCTIONS: PROCESS_CLIENT, PROCESS_PAGER, PRINT_PAGER

- process_client():
 - reads each pager for this client number from the pager database table (mdspager).
 - calls print_pager() to read the pager’s stats (from the mdspasts table) and write them to the report
- process_pager():
 - cycles through the list of pager numbers entered, checking each against mdspager
 - calls print_pager() to read the pager’s stats (from the mdspasts table) and write them to the report
- print_pager():
 - uses n_read_rec() (coded in mdsrsrtn.c) to read the pager stats from the <System_Name> network, the n_*_rec() functions being for the reading of databases at arbitrary <System_Name> sites.
 - Does some checks, some formatting, and (actually does) writes the stats to the report

20.6 DETAILED LINE REPORT: SKYMD547 (AND SKYMD547A)

coded in:	skymds47.c
called from (code):	report_menu() in skymds00.c
called from (window):	Base <System_Name> menu, report sub-menu, “Detailed Line Report” selected from sub-menu.
raison d’ etre:	Produce a report detailing <System_Name>’s setup for the indial (“Line Number”) supplied, with a couple of options for detailed ‘contact’ (“Advanced Processing Services”) listings. The program skymds47a is run to generate the report.
screen:	
database tables:	mdsports
sub-program:	All of them (let’s say).

20.6.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_report() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - open the line database table, mdpline
- select_report() calls display_screen() to finish constructing, and display, the “Detailed Line Report” window.

20.6.2 FUNCTION: SELECT_REPORT

select_report either:

- calls email_report(), which runs the script **\$MDS_BIN_DIR/dlr.script** to generate and email a report, or
- calls spawn_report(), which runs skymds46a with the indial and report options as command-line parameters.

20.6.3 SCRIPT: DLR.SCRIPT (CALLED BY EMAIL_REPORT())

dlr.script, in the directory **\$MDS_BIN_DIR/** merely:

- runs skymds47a with the report generation options as command-line parameters
- if the resulting report is greater than 2MB, it is compressed.
- finally the report is emailed.

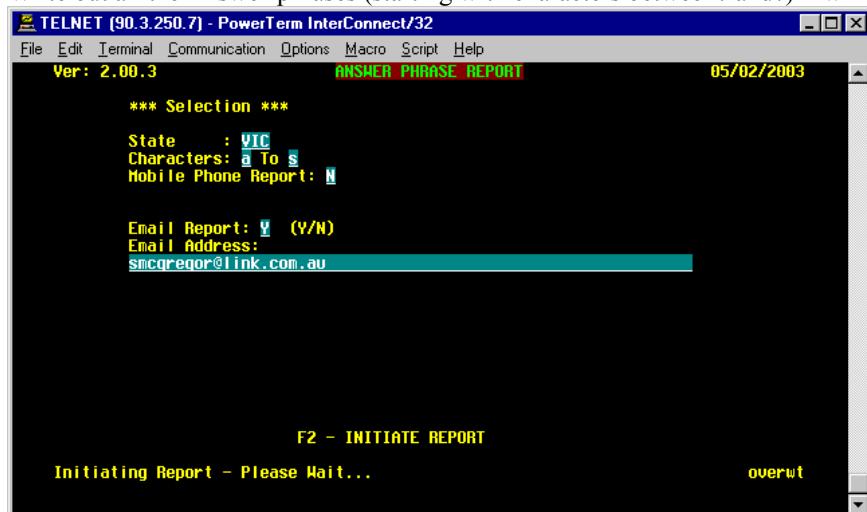
20.6.4 PROGRAM: SKYMD547A

skymds47a consists of three main parts:

- line_details() adds information on the line setup to the report
- process_structure() lists ‘one level’ of contacts, but is calls itself recursively to detail all the clients contacts.
- records_report() calls a specialised printing function for each contact type. This function det_records() (e.g. adv_records()) calls process_contact, which calls the **real** specialised function, get_recs(). It is this function that formats and writes the contact data from the process_structure() call in the previous step. In some cases the get_det_recs() function gathers the data itself as well.

20.7 ANSWER-PHRASE REPORT: SKYMDS48 (AND SKYMD548A)

coded in: skymds48.c
called from (code): report_menu() in skymds00.c
called from (window): Base <System_Name> menu, sub-menu selected, “Answer Phrase Report” selected from sub-menu.
raison d’ etre: Write out all the Answer-phrases (starting with characters between? and?) - will be big.
screen:



database tables:

mdsconta
mdspager

20.7.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then select_report() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the database tables
- select_report() calls display_screen() to finish constructing and display the “Answer Phrase Report” window.

20.7.2 FUNCTION: SELECT_REPORT AND SPAWN_REPORT

select_report()

- calls display_screen() to display the “Answer Phrase Report” screen.
- waits for an F2 key-press, and call spawn_report() when it gets one

spawn_report():

- calls validate_screen() to check that the options are entered and make sense.
- prepare the printer or the (temporary) email file
- call spawn skymds48a to generate the report. The options are passed as command-line parameters.
- if emailing, use sendmail to email the report.

20.7.3 PROGRAM: SKYMD548A

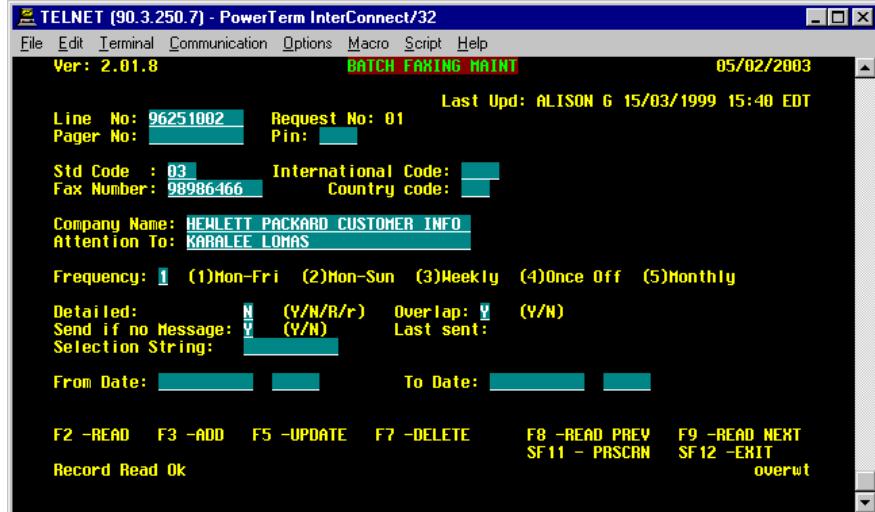
skymds48a uses its gen_report() function to produce the Answer-Phrase report. gen_report() uses three functions to do this:

- store_report(): goes through the line database table (mdsline), retrieving all the answer-phrases for the selected state that begin with the stated letter and store them in a file.
- sort_report(): uses the Unix operating system ‘sort’ utility to sort the records into another file in alphabetical order. Better than generating another index solely for this seldom used utility.
- print_report(): repeatedly call print_record() to format the answer-phrase, writing them a third time to a third file.

Once skymds48a returns, spawn_report() either emails or prints the file just produced.

20.8 “BATCH FAX SETUP”: SKYMDS19

coded in: skymds19.c
called from (code): report_menu() in skymds00.c
called from (window): Base <System_Name> menu, report sub-menu, “Batch Fax Setup” selected from sub-menu.
raison d’ etre: Set-up batch faxes, where messages are stored and sent in a single “Batch” fax.
screen:



database tables:

mdsfax
 mdstimev
 mdsline
 mdspager
 mdspassw
 mdsaudit
 mdsports

20.8.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information.
 - Test that the user is authorised to send batch faxes.
- process_screen() calls display_screen() to finish constructing and display the “Batch Fax Setup” window.

20.8.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

- calls display_screen() to re-display the Batch Fax data, possibly updated
- uses edit_scrn() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_fax()	read the next Batch Fax from the mdsfax table
F3	add_fax()	check (validate_screen()), add a Batch Fax record, copy to maintenance queue
F5	update_fax()	save changes to the current Batch Fax and write an audit trail record.
F7	delete_fax()	delete the current Batch Fax on all <System_Name> servers (via the maintenance queue)
F8	prev_fax()	move to the previous Batch Fax in the database.
F9	next_fax()	move to the next Batch Fax in the database.
also	validate_screen()	checks that the fax details on the screen make sense (before adding or updating). <ul style="list-style-type: none"> - validate line number and pager number, - checks security levels, - checks and optionally modifies the fax number, - a company name must be entered, - appropriate fax frequency, - date and time entered (properly)

20.9 “BATCH FAX LIST”: SKYMD546E

coded in:

skymds46e.c

called from (code):

report_menu() in skymds00.c

called from (window):

Base <System_Name> menu, report sub-menu, “Batch Fax List” selected from sub-menu.

raison d’ etre:

Produce a text-file report of all the Batch Faxes set up in <System_Name>, batch faxes

screen:



database tables:

mdstimev

mdsfax

20.9.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() to set-up the program. Other than the list of cities to choose from there is no user interface to construct.
- initialise()
 - reads in the environment variables it needs
 - parses the command-line
 - catches some signals
 - open the timed events (mdstimev) and fax (mdsfax) database tables
 - use openprint() (code in mdsinit.c) to open a pipe to the printer named by the MDS_PRINTER environment variable.

20.9.2 FUNCTION: PRINT_HEADINGS

A simple function that prints the column headers and separation lines ('-----') for the Batch Fax report

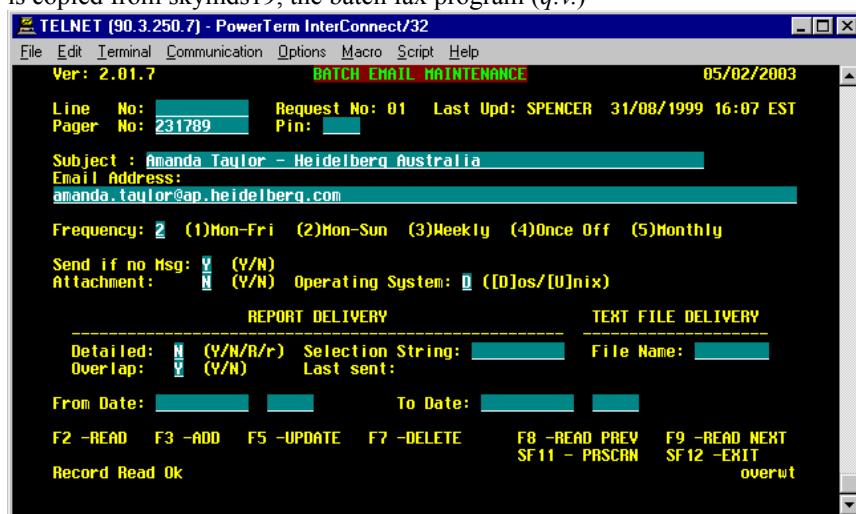
20.9.3 FUNCTION: PROCESS_BATCH_FAX

- uses load_fax_recs() to read the batch fax records from the mdsfax database table into memory
- the Unix operating system utility ‘qsort’ sorts the records in memory
- finally each fax record is printed by an individual call to print_fax(), which both formats and prints.

20.10 “BATCH EMAIL SETUP”: SKYMDS19A

coded in: skymds19a.c
called from (code): report_menu() in skymds00.c
called from (window): Base <System_Name> menu, report sub-menu, “Batch Email Setup” selected from sub-menu.
raison d’ etre: Set up “Batch” emails, where messages are stored and sent in a single “Batch” email. This is copied from skymds19, the batch fax program (q.v.)

screen:



database tables:

mdsemail
mdstimev
mdspager
mdstext
mdspassw
mdsaudit
mdsports

20.10.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - open the database tables
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - connects to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information.
- process_screen() then calls display_screen() to display the “Batch Email Setup” screen.

20.10.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

- calls display_screen() to re-display the Batch Email data, possibly updated
- uses edit_sern() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_email()	read the next Batch Email record. Gets its start-stop dates from the timed events table
F3	add_email()	<ul style="list-style-type: none"> - check the user input (validate_screen()) - use load_email_rec() to read the email from the mdsemail database table - send the “add email” with details to the maintenance queue - write an audit record - if dated (start – stop) add to the timed events table
F7	delete_email()	delete batch email, write to the maintenance queue, and write and Audit Record
F8	prev_email()	move to the previous Batch Email in the database.
F9	next_email()	move to the next Batch Email in the database.
also:	validate_screen()	checks that the email details on the screen make sense (before adding or updating). (See the validate_screen() in skymds19: Batch Fax Set-up)
	read_timed_event()	Batch Emails that have a Start and a Stop date have their start and stop dates added to the timed events table, so the initiation and removal of the service will / does happen automatically.
	add_timed_event()	
	del_timed_event()	

21 Customer Menu

<System_Name> staff have developed a number of custom solutions for the benefit of ‘special’ <Company_Name> customers, these custom solutions implemented as “external databases”, of which four remain in <System_Name>:

1. V.I.P. <Company_2_Name> call handling
2. <Company_1_Name> (cars) service request management
3. <Company_3_Name> Facilities Response Centre (FRC),
4. Safety <Company_Name>, a Personal Safety monitoring system,

The “Customer” menu, implemented as skymds00’s customer_menu() function, provides interfaces to the maintenance of each of these four “external databases” custom solutions. customer_menu() operates in the same general manner as the other *_menu() functions:

- the menu is first constructed, the user’s security level being tested before each item is added.
- Next the help context is added. Unfortunately the help for the reports menu is displayed instead of any help for this custom solution menu.
- Now the “Customer” menu is displayed.

Menu item	Calls:	Description
Trade/Service Code Maint	skymds30.c	Prepare codes for a new custom solution.
V.I.P. / V.I.P. Menu <Company_1_Name>/X Menu FRC (<Company_3_Name>)	vip_menu() startel_menu() frc_menu()	Home maintenance company set-ups <Company_1_Name> dealer information <Company_3_Name> company info
Safety <Company_Name> Menu	saf_menu()	Employee (safety) monitoring set-ups

Table 110: skymds00: Customer Menu actions

- rev_select() reads the user’s menu selection.
- The menu-item text is then read to determine which sub-menu function (or skymds30) to call.

21.1 CUSTOMER MENU: SUB-MENUS

Each of these sub-menus calls the appropriate sub-menu function: vip_menu(), startel_menu(), frc_menu(), saf_menu(), and each of these functions is written, and operates, in the same way as each other, and the top-level menus.

21.1.1 SUB-MENU FUNCTIONALITY

- the menu is constructed, the user’s security level being tested before each item is added.
- the (incorrect) help context is set
- The sub-menu is displayed.
- The function, say frc_menu(), now polls for the user’s selection
- rev_select() reads the user’s selection, the sub-menu function calling the appropriate sub-program (see the following tables).

The sub-menu program is spawned.

- PORT_update_port_rec() is used update the details in the mdsports database table to indicate that the current running process is no-longer whatever was spawned, but (back to) skymds00 again.
- Once the sub-menu exits the memory allocated for the menu items is freed.

21.1.2 “V.I.P.” MENU

The VIP solutions have been developed for the <Company 2 Name> companies. The “V.I.P.” Menu runs programs as per the following table.

Menu item	Program	Description
VIP Bulk Data Entry	cusvip02	Add delete new areas and services to the VIP Company
VIP Database Maintenance	cusvip03	Read and change VIP data held by <Company_Name>
VIP Database Report	cusvip06	Report based on entered data.

Table 111: Customer Menu: V.I.P. Sub-menu

21.1.3 “<COMPANY_1_NAME>/X” MENU

This menu maintains <Company_1_Name> (<Company_1_Name> X) dealer location information for Australia, X being the previous name of this service. The “<Company_1_Name>/X” Menu runs programs as per the following table.

Menu item	Program	Description
<Company_1_Name>/X Dealer Maint	cussta02	Maintain the <Company_1_Name> dealer info
<Company_1_Name>/X Location Maint	cussta04	Simpler where-is-each-dealer info

Table 112: Customer Menu: <Company_1_Name>/X Sub-menu

21.1.4 “F.R.C.” MENU <COMPANY_3_NAME>)

The F.R.C. Menu provides maintenance functionality for the range of specialised services developed for the <Company_3_Name> Retail network.

Menu item	Program	Description
FRC Manager Maint	cusfrcfm	Maintain <Company_3_Name> manager info
FRC Enquiry Type Maint	cusfrct	Maintain Enquiry Type info
FRC Enquiry Code Maint	cusfrcen	Maintain Enquiry Code info
FRC Site Maint	cusfrcsi	Maintain Site info
FRC Enquiry/Site Maint	cusfrces	The relationship between location and enquiries
FRC Trade Code Maint	cusfrctr	Maintain trade codes
FRC Maintenance Code Maint	cusfrcma	Maintain maintenance codes.
FRC Service Provider Maint	cusfrrsp	Maintain service providers
FRC Trade/Site Maint	cusfrcts	Maintain the Trade code – location relationships
FRC Site/Cost Centre Maint	cusfrcsc	Maintain the location – cost centre relationships
FRC Escalation Rebuild	cusfrcer	Build message escalation rules.

Table 113: Customer Menu: F.R.C. Sub-menu

21.1.5 “SAFETY <COMPANY_NAME>” MENU

“Safety <Company_Name>” is a service provided by us (i.e. <Company_Name>) enabling organisations to monitor the safety of their staff in dangerous environments. The staff call in periodically to acknowledge that they’re OK. If no call is received in a specified time a ‘duress’ and an alarm are raised.

Menu item	Program	Description
Safety <Company_Name>	cussafpr	Information about the people being monitored
Profile Screen		
Safety <Company_Name> Service File	cussafsp	Prompts and set-up for the Safety <Company_Name> service
Safety <Company_Name> log file Enquiry	cussaflg	Search past Safety-<Company_Name> data
Report Generator	cussafrp	Produce a report on Safety <Company_Name> actions.

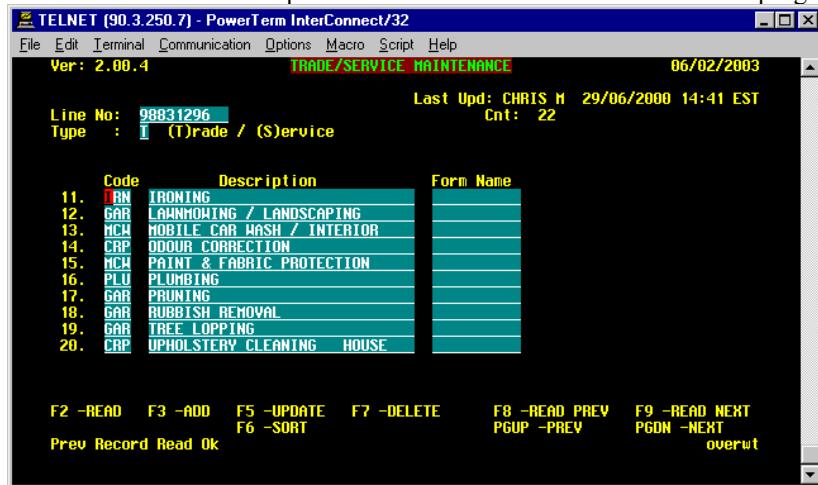
Table 114: Customer Menu: Safety-<Company_Name> Sub-menu

21.2 CUSTOMER MENU: TRADE/SERVICE CODE MAINT: SKYMD30

coded in: skymds30.c
called from (code): report_menu() in skymds00.c
called from (window): Base <System_Name> menu, Customer sub-menu, “Trade/Service Code Maint” selected
raison d’ etre: Maintain codes and descriptions for each of the “Customer” customer programs .
screen:

database tables:

mdsline
 mdsscode
 mdsparts



21.2.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the database tables and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
- process_screen() calls display_screen() to construct and display the “Trade/Service Code Maint” window.

The screen appears during the first display_screen() call at the start of process_screen().

21.2.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

- calls display_screen() to re-display the possibly updated, or otherwise new, Trade/Service Code data.
- uses edit_scren() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_codes()	read the next Trade/Service Code record from the mdsscode table
F3	add_codes()	use insert_code() to add a Trade/Service Code. Call validate_screen() to check.
F5	update_codes()	Call validate_screen(), save changes, and copy to the maintenance queue.
F7	delete_codes()	use remove_codes() to delete the current Trade/Service Code record.
F8	prev_codes()	move to the previous Trade/Service Code record in the database.
F9	next_codes()	move to the next Trade/Service Code recording the database.
Page-up	prev_page()	Scroll up to the previous page of Trade/Service Codes for the current client
Page-down	next_page()	Scroll down to the next page of Trade/Service Codes for the current client

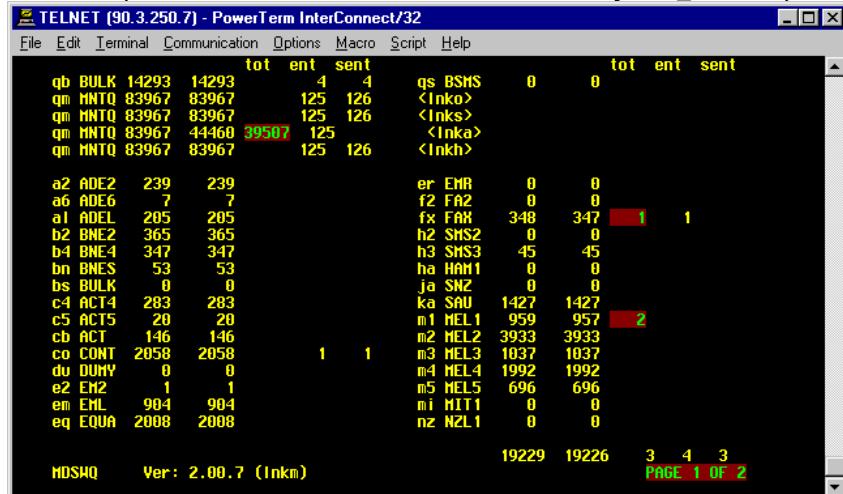
22 Utilities Menu

Menu Item	Program	Description
Watch Queue (60 Seconds)	mdswq	Monitor all <System_Name> queues – update every minute
Watch Queue (5 Seconds)	mdswq	Monitor all <System_Name> queues – update every 5 seconds
Operator List Report	skymds33	List Call Centre Operators by city and security group
Terminal Monitoring	mdstymon	remotely monitor an operator's terminal
Network Delay Monitoring	skymds78	view faxes sent by indial, fax number, or (original) pager number.
Network Connection maintenance	skymds79	Manage the (many) <System_Name> network connections.
Mobile Phone Maintenance	skymds65	Special information for special (e.g. Virgin / Optus) mobiles
Port Maintenance	skymds84	Static, fixed information on ports
Alarm Maintenance	skymds92	Set the testing frequency and destination of <System_Name> alarms.
Customer Program Maintenance	skymds88	When to run the custom solutions. (see 'customer' menu)
Fax Delivery Status	skymds78	Monitor, which faxes will be and have been sent.

Table 115: skymds00: Utilities Menu actions

22.1 WATCH QUEUE (60 AND 5 SECONDS): MDSWQ

coded in: mdsdq.c
called from (code): utilities_menu() in skymds00.c
called from (window): Base <System_Name> menu, Utilities sub-menu selected, "Watch Queue (60, 5 Seconds)" selected.
raison d' etre: Allow a supervisor or administrator to observe all the <System_Name> queues.
screen:



22.1.1 PROGRAM & SCREEN INITIALISATION

- initialise() prepares the program and calls get_dests() and display_dests() to prepare the screen.
 - reads in the environment variables it needs
 - parses the command-line
 - catches some signals
 - opens and prepares the communications ports it needs
 - open the destinations (queues) database table (mdsdestm)
 - initialises the curses system
- get_dests() now reads each queue from the "destinations" file mdsdestm, and writes the queue statistics to the curses screen about to be delayed.

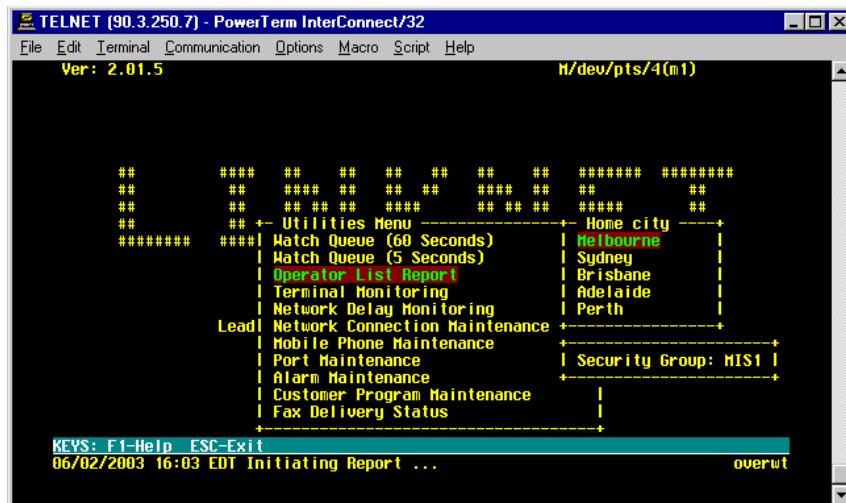
22.1.2 FUNCTIONS: MAIN AND PROCESS_DESTS

- main() runs in a loop,
- calls process_dests() to read the queue traffic
 - sleeps for 60 seconds, or 5 seconds, or till the user presses a key
 - scrolls in response to page up/down, Ctrl-f/b key-presses
 - display_dests() updates the column headings

22.2 “OPERATOR LIST REPORT”: SKYMDS33

coded in: skymds33.c
called from (code): utilities_menu() in skymds00.c
called from (window): Base <System_Name> menu, Utilities sub-menu, “Operator List Report” selected from sub-menu.
raison d’ etre: Print a report of each operator in the security group, plus their call statistics (for the year).

screen:



database tables:

mdspassw
mdsports

22.2.1 PROGRAM INITIALISATION

22.2.1.1 In skymds00's utility_menu() function

NB: this is before skymds33 has been run.

- the “Home City” menu is displayed by skymds00 calling select_home_city().
- Next the “Security Group” dialog / text box is shown into which the user must enter something.

Now skymds33 is run with command-line parameters gleaned from the answers above.

22.2.1.2 skymds33's initialise() function

initialise():

- reads in the environment variables it needs
- open the password database table
- parses the command-line.
- catches some signals
- prepares the print environment

There is no curses / screen environment associated with skymds33.

22.2.2 REPORT GENERATION: FUNCTION PROCESS_PRINT

process_print() generates the operator report.

- first print_header() is used to construct the column headings.
- Now the password database table is traversed (indexed by city)
- If the operator belongs to the chosen security group, print_oper() is called to add them to the security report.

22.2.3 FUNCTION: PRINT_OPER

writes the

- Name, ID, Phone number / contact, Security Group
- Calls per year, and characters typed per year

to the operator report.

22.3 “TERMINAL MONITORING”: MDSTYMON

coded in: mdstymon.c
called from (code): utilities_menu() in skymds00.c
called from (window): Base <System_Name> menu, Utilities sub-menu, “Terminal Monitoring” selected from sub-menu.

raison d’ etre: Watch a Call Centre Operator at work, observing all their key strokes, screens, etc.

screen:



database tables:

mdsports

22.3.1 PROGRAM & SCREEN INITIALISATION

initialise() initialises mdstymon and prepares the (local) curses environment, which displays a duplicate of the monitored Call Centre Operator’s screen. initialise() is the same as usual:

- catches some signals
- initialises the curses system
- open the ports database table (mdsports)
- construct a pointer to the security shared memory

22.3.2 MONITOR THE OPERATOR: FUNCTIONS: TTYSYMON_SELECT AND TURN_ECHO_ON

22.3.2.1 Function: ttymon_select

ttymon_select reads generates and displays the list of operators (as shown above) for you, good reader, to choose from.

- first get_arr_oper()
 - reads through the ports shared memory
 - fills out an array with info on each port, flagging each of those that are active
- writes the array out to (NB:) stderr, which must be caught by the curses system somewhere.
- if an operator is selected by the user, and return hit:
 - the ‘device’ (`/dev/some_tty_terminal`, i.e. port) is determined
 - turn_echo_on() starts monitoring.
- alternatively, F12, indicating that the sorting order should be changed

22.3.2.2 turn_echo_on

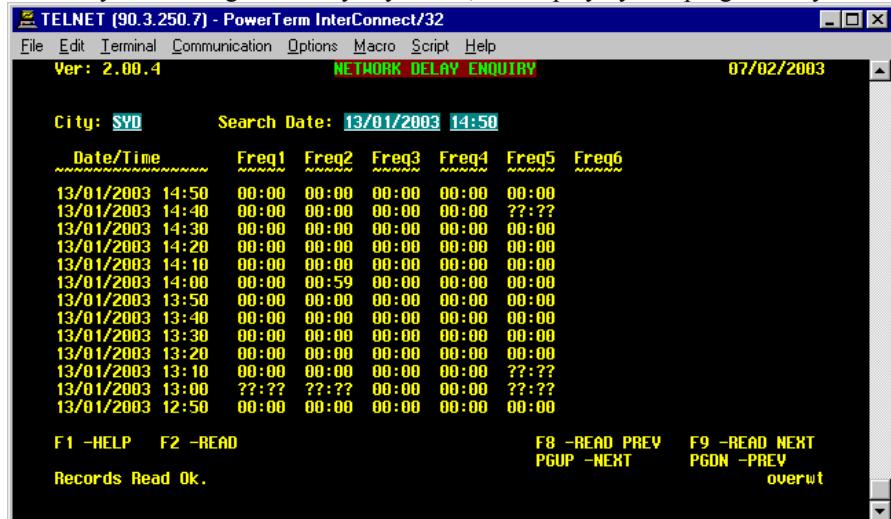
- the utility find_rec_for_port() returns a pointer to (the correct part of) the ports’ shared memory
- The PID of the ‘target’ operator’s application (skymds00?) is obtained
- The operator’s terminal is flagged as being monitored in the database.
- The ports’ shared memory record of the operator has our terminal (port/device, e.g. `/dev/our_term_tty`) written into the operators’ ‘echo_device’ field.

It is this last step, our terminal being written into the operators’ echo_device field that makes a duplicate of their (the operator’s) terminal appear on our screen.

22.4 “NETWORK DELAY MONITORING”: SKYMDS83C

coded in: skymds83c.c
called from (code): utilities_menu() in skymds00.c
called from (window): Base <System_Name> menu, Utilities sub-menu, “Network Delay Monitoring” selected.
raison d’ etre: Enable users to see network response times, by state and frequency, in 10 minute intervals. The delay monitoring is done by skymds83, the display by this program: skymds83c .

screen:



database tables:

mdsports
mdsdelay

22.4.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - open the network delay database table (mdsdelay)
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure

The screen first appears during the first display_screen() call at the start of process_screen().

22.4.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

- calls display_screen() to re-display the Network Delay data, which will change after each F8/F9 and page-up/down
- uses edit_scrn() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

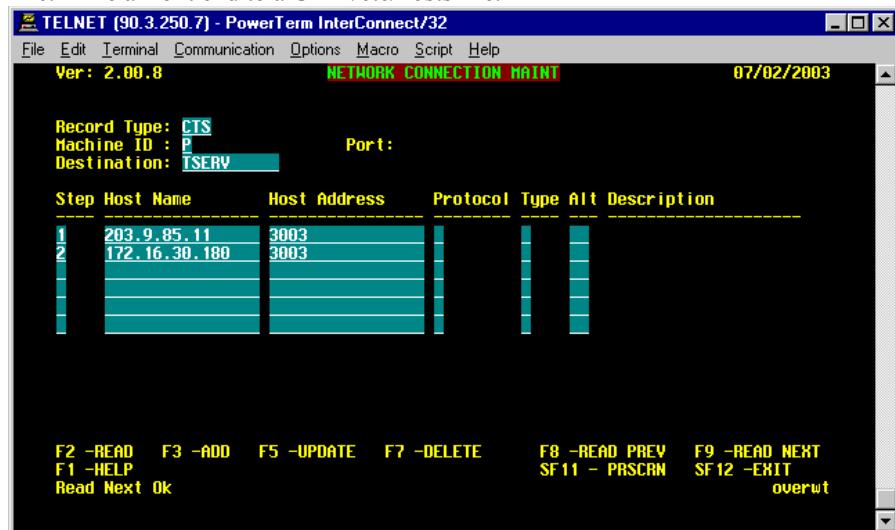
Key	Function Called	Description
F2	read_delay()	read the appropriate Network Delay records from the mdsdelay table
F8	prev_delay()	move to the previous Network Delay record (previous day, same time)
F9	next_delay()	move to the next Network Delay record (next day, same time) in the database.
Page-Up	next_page()	scroll up the network delays, showing earlier timed delay tests.
Page-Down	prev_page()	scroll down the network delays, showing later timed delay tests.

NB: all the Network Delay records are generated and stored by the skymds83 daemon, skymds83c merely displays existing records from the mdsdelay table.

22.5 “NETWORK CONNECTION MAINTENANCE”: SKYMDS79

coded in: skymds79.c
called from (code): utilities_menu() in skymds00.c
called from (window): Base <System_Name> menu, Utilities sub-menu selected, “Network Connection Maintenance” selected from sub-menu.
raison d’ etre: View and maintain the network setup information, IP addresses, ports, hostnames and the like. Like a front end to a Unix /etc/hosts file.

screen:



database tables:

mdsconta
mdspager

22.5.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - open the database tables
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open a connection to the maintenance queue (for inter-site updates)
 - construct a pointer to the location of this user’s shared memory security information

The screen appears during the first display_screen() call at the start of process_screen().

22.5.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

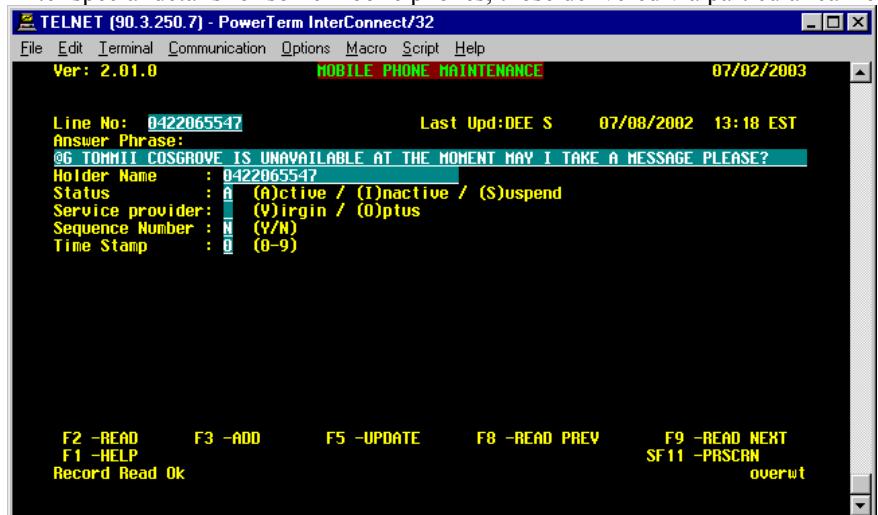
- calls display_screen() to re-display the Network Connection record data, possibly updated
- uses edit_scren() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_netcn()	read the next Network Connection record from the mdsDBDB table
F3	add_netcn()	check and add a Network Connection record. See validate*() functions below.
F5	update_netcn()	save changes to the current Network Connection record. See validate*() functions.
F7	delete_netcn()	delete the current Network Connection record
F8	prev_netcn()	move to the previous Network Connection record in the database.
F9	next_netcn()	move to the next Network Connection record in the database.
and	validate_header() validate_detail()	Check that a records type and a Machine ID is entered for any additions or changes Check all the other fields on the screen before saving.

Happiness and Love.

22.6 “MOBILE PHONE MAINTENANCE”: SKYMDS65

coded in: skymds65.c
called from (code): utilities_menu() in skymds00.c
called from (window): Base <System_Name> menu > utilities sub-menu > “Mobile Phone Maintenance”
raison d’ etre: Enter special details for some mobile phones, those delivered via particular carriers.
screen:



database tables:

mdsline
mdspager
mdsports

22.6.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs.
 - opens its database tables.
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - opens a connection to the maintenance queue.
 - connects to the security shared-memory, to check that the current user has privilege to run this program.

The screen appears during the first display_screen() call at the start of process_screen().

22.6.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

- calls display_screen() to re-display the Mobile Phone (line) record data, possibly updated
- uses edit_scrn() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

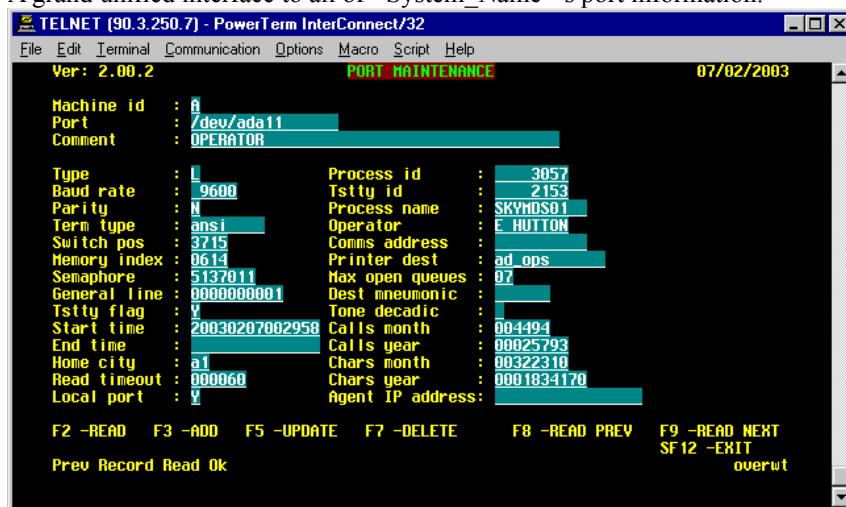
Key	Function Called	Description
F2	read_rec()	read the next Mobile Phone (line) record and associated data (from mdspager)
F3	add_rec()	check (check_rec())and add a Mobile Phone (line) record.
F5	update_rec()	save the current Mobile Phone record and update it to the maintenance queue.
F8	read_prev()	move to the previous Mobile Phone (line) record in the database.
F9	read_next()	move to the next Mobile Phone (line) record in the database.
also	validate_screen() setup_pager_rec() setup_line_rec()	check that all the fields are correctly filled Prepare a record for the mdspager table, containing the entered mobile phone data. ditto for the mdsline table.

22.7 “PORT MAINTENANCE”: SKYMD84

coded in: skymds84.c
called from (code): utilities_menu() in skymds00.c
called from (window): Base <System_Name> menu, Utilities sub-menu, “Port Maintenance” selected from sub-menu.

raison d’ etre: A grand unified interface to all of <System_Name>’s port information.

screen:



database tables:
mdsports

22.7.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - open the ports database table (mdsports) and a connection to the maintenance queue
 - construct a pointer to the location of this user’s shared memory security information
 - If the user is not a supervisor, then limit updating rights

The screen appears during the first display_screen() call at the start of process_screen().

22.7.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

- calls display_screen() to re-display the port record data, possibly updated
- uses edit_scren() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_port()	read the next port record from the mdsparts table
F3	add_port()	check (validate_data()), add a port record, and copy to the maintenance queue.
F5	update_port()	save changes to the current port record and update to the maintenance queue.
F7	delete_port()	delete the current port record
F8	prev_port()	move to the previous port record in the database.
F9	next_port()	move to the next port record in the database.
also	validate_data() load_screen()	a few checks to ensure that the data entered is correct after each update, load_screen() redisplays the data. display_screen() is also called

22.8 “ALARM MAINTENANCE”: SKYMD92

coded in:	skymds92.c
called from (code):	utilities_menu() in skymds00.c
called from (window):	Base <System_Name> menu, Utilities sub-menu, “Alarm Maintenance” selected from sub-menu.
raison d’ etre:	Enable, disable, and determine the handling of the <System_Name> alarms.
screen:	
database tables:	mdsalarm mdspager mdsports

22.8.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - open the database tables
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - construct a pointer to the location of this user’s shared memory security information
 - initiates a connection to the maintenance queue.

The screen appears during the first display_screen() call at the start of process_screen().

22.8.2 FUNCTION: PROCESS_SCREEN

After performing an initial clear-out of the display fields, process_screen(), in an infinite loop:

- calls display_screen() to (re-)display the alarm record data, possibly updated
- uses edit_scrn() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

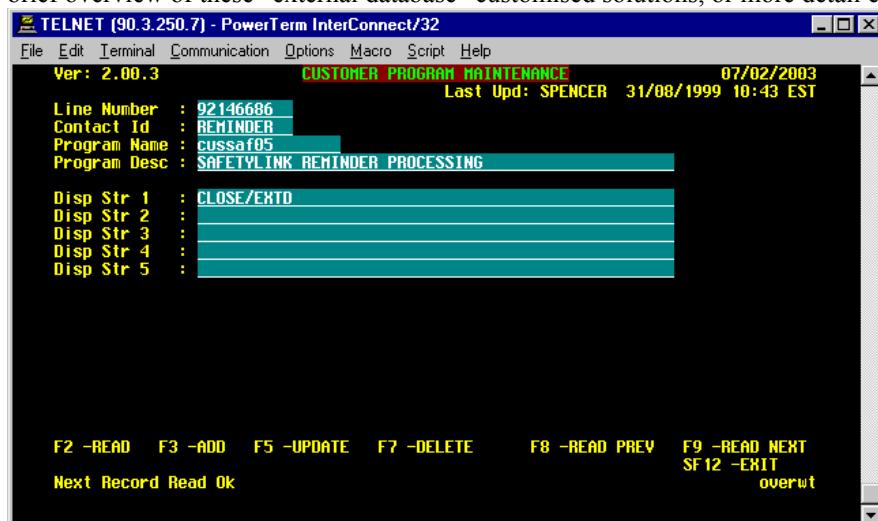
Key	Function Called	Description
F2	read_alarm()	read the next alarm record from the mdsDBDB table
F3	add_alarm()	check and add a alarm record.
F5	update_alarm()	save changes to the current alarm record and update to the maintenance queue.
F7	delete_alarm()	delete the current alarm record
F8	prev_alarm()	move to the previous alarm record in the database.
F9	next_alarm()	move to the next alarm record in the database.
also	validate_data()	ensure that the fields have been properly filled in

22.9 “CUSTOMER PROGRAM MAINTENANCE”: SKYMD88

coded in: skymds88.c
called from (code): utilities_menu() in skymds00.c
called from (window): Base <System_Name> menu, Utilities sub-menu, “Customer Program Maintenance” selected.

raison d’etre: Attach certain indials to customised solutions. See the section on the Customer menu for a brief overview of these “external database” customised solutions, or more detail elsewhere

screen:



database tables:

mdscustp
mdsconta
mdsports

22.9.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise():
 - catches some signals
 - reads in the environment variables it needs
 - open its database tables
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - construct a pointer to the location of this user’s shared memory security information
 - initialises a connection to the maintenance queue

The screen appears during the first display_screen() call at the start of process_screen().

22.9.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

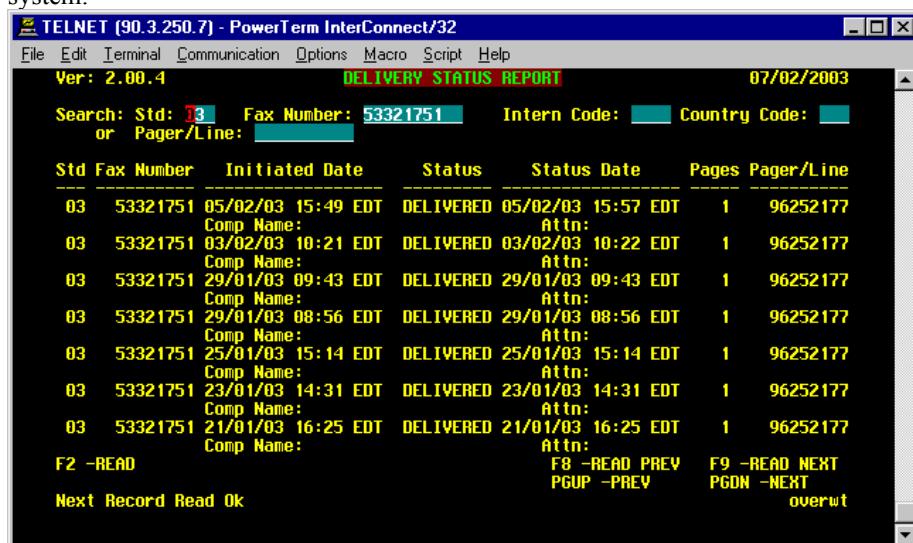
- calls display_screen() to re-display the ‘customer program’ record data, possibly updated
- uses edit_scren() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_custp()	read the next ‘customer program’ record from the mdsDBDB table
F3	add_custp()	check and add a ‘customer program’ record.
F5	update_custp()	save changes to the ‘customer program’ record and update to the maintenance queue.
F7	delete_custp()	delete the current ‘customer program’ record
F8	prev_custp()	move to the previous ‘customer program’ record in the database.
F9	next_custp()	move to the next ‘customer program’ record in the database.
also	validate_data()	before doing an ‘add’ or an ‘update’, ensure that the custom service being <Company_Name>ed to the indial actually exists

22.10 “FAX DELIVERY STATUS”: SKYMDS78

coded in: skymds78.c
called from (code): utilities_menu() in skymds00.c
called from (window): Base <System_Name> menu, utilities sub-menu, “Fax Delivery Status” selected from sub-menu.
raison d’ etre: Monitor the “DELIVERED”, or otherwise, status of faxes sent from the <System_Name> system.

screen:



database tables:

mdsdsrec
mdsline
mdspager
mdspassw
mdsports

22.10.1 PROGRAM & SCREEN INITIALISATION

- main() calls initialise() and then process_screen() to display the screen and manage user interactions.
- initialise()
 - catches some signals
 - reads in the environment variables it needs
 - opens database tables
 - initialises the curses system and writes the screen layout to the mainwin “WINDOW” structure
 - construct a pointer to the location of this user’s shared memory security information

The screen appears during the first display_screen() call at the start of process_screen().

22.10.2 FUNCTION: PROCESS_SCREEN

In an infinite loop, process_screen():

- calls display_screen() to re-display the Fax Delivery Status record data, possibly updated
- uses edit_scm() to retrieve the user’s key press. If it is a function key, then a function is called as per the table:

Key	Function Called	Description
F2	read_deliv_stat_rec()	read the next Fax Delivery Status record from the mdsDBDB table
F8	prev_deliv_stat_rec()	move to the previous Fax Delivery Status record in the database.
F9	next_deliv_stat_rec()	move to the next Fax Delivery Status record in the database.
Page-Up	next_page()	scroll up the list of Fax Delivery Status records.
Page-Down	prev_page()	scroll down the list of Fax Delivery Status records.

23 skymds20

23.1 SKYMDS20 OVERVIEW

skymds20 extracts <System_Name> messages from a bulk queue and assigns them to the correct copy of skymds03. A running <System_Name> system will have 2 copies of skymds20: a Bulk SMS skymds20, and an ‘everything-else’ skymds20. The former directs messages only to the SMSC destination queue, while the later delivers each message to the correct instance of the 30 or so copies of skymds03.

23.2 RUNNING SKYMDS20

Like most other <System_Name> programs, skymds20 is run from the start.mds <System_Name> start-up script residing in MDS_BIN_DIR (usually /opt/mds/bin).

During the processing of the start.mds script file, the user is prompted, basically: Do you want to start the <System_Name> daemons?

```
Start skymds11, skymds67, skymds18, skymds25, skymds28, skymds81, skymds06,  
skymds27, skymds10, remterm and skymds20 (y/n) :
```

if the user types in ‘y’, the programs listed, including our skymds20, are run.

23.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-a	Process Bulk SMS. Bulk SMS uses an <u>alternative</u> Bulk Queue.
-t < port >	Dummy communications port for indexing into the mdsports database table
-D	Run in debug mode, listing detailed run-time information.
-p < queue >	Source (2 nd) queue of a dual queue setup – all messages to this queue also go to the ‘destination’
-c < queue >	Destination (2 nd) queue of a dual queue setup. Receive duplicates of the ‘source’ queue messages
-s	Do not send messages from pre-set ‘test’ operators to the destination queues (for testing)
-o < operator >	Do not send messages from this operator to the destination queues (for testing)
-i < pager num >	Do not send messages from this pager to the destination queues (for testing)
-?	print usage instructions to stdout.

Table 116: skymds20: Command-Line Parameters

23.2.2 EXAMPLE COMMAND-LINE

At one <System_Name> site, the start.mds start-up script uses the following two command-lines to run a normal and a Bulk-SMS skymds20:

```
$MDS_BIN_DIR/skymds20 -t /dev/dum04 2>$LOG_SCR  
and  
$MDS_BIN_DIR/skymds20 -a -t /dev/dum05 2>$LOG_SCR
```

indicating that skymds20, found in the MDS_BIN_DIR directory, is to run:

- tagged with the communications port /dev/dum04 or /dev/dum05
- the first is processing ‘normal’ bulk queue messages, the second processing the ‘alternative’ Bulk SMS queue.
- the stderr file is duplicated to the LOG_SCR device, a terminal somewhere.

23.3 PROGRAM DESCRIPTION

As described above, skymds20 distributes messages from the/a Bulk Queue for delivery to an appropriate destination queue. Which Bulk Queue skymds20 reads from is indicated by the presence or absence of the -a command-line parameter, while the destination queue for each message is found by querying the mdsdests (destination queue) database table for each message.

23.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds20.

23.4.1 SOURCES FILES

Source File	Description
skymds20.c	Extract messages from the bulk queue (mdsqa0.txt)and put them in the real queues.
alarmrtn.c	Generic alarm routine.
gensubs.c	Contains various general-purpose routines that are separately compiled.
globvar.c	Functions to access the Global Variable shared memory used by multiple programs.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
mdscdr.c	Contains various general-purpose routines that are separately compiled.
mdsemqmain.c	Generic message email queuing routine.
mdsginit.c	A number of functions, which perform commonly, used initialisation code segments.
mdshisio.c	Generic history file handling routines.
mdskbld.c	Contains functions to build and unbuild pager keys, and setup a dummy pager.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the ports file.
mdspstats.c	Implement <code>add_pager_stats()</code> to record <System_Name> message and program statistics
mdsqall.c	Generic message queuing routine.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurtn.c	Generic message queuing routine.
mdssetup.c	Obtain setup definitions from a text setup file.
mdstmrtn.c	Contains functions, which perform operations to do with time.

Table 117: skymds20: Source Files

23.4.2 HEADER FILES

Header File	Description
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.: C time field definitions
macros.h	User defined macros.: definitions & macros for common functions
mdsacrec.h	Alternate CDR file
mdsaprec.h	A-Party Master file
mdscdr.h	CDR definitions file
mdsderec.h	Ports security master file
mdsfarec.h	Pager facilities master file
mdsflsz.h	Standard <System_Name> includes.: field len & definitions
mdshirec.h	History primary recs file: New History logging file
mdshsrec.h	Secondary History file (audits)
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsparam.h	parameters used by PMS system
mdsparec.h	Pager Master File
mdsporec.h	Ports security master file: Ports Record
mdsqarec.h	Encoder queue: Message queue file
mdsqdrec.h	Detailed bulkq
mdsquirec.h	Contacts Message queue file: Destination Message queue file
mdssqrec.h	Message sequence number file
mdsstrec.h	Pager stats master file
mdsterec.h	Timed event file (incl. booked call)

Table 118: skymds20: Header Files

23.4.3 DATABASE TABLES

Each of these header files contains (only) a C-struct that matched the structure of a database table used by skymds20.

Database Table	Header File	Description
hiccyymmdd	mdshirec.h	the 'logical' history
hsyymmdd	mdshsrec.h	History secondary storage
mdsacrec	mdsacrec.h	Alternative call details master
mdsaparty	mdsaprec.h	A-Party Customer Info
mdsdestm	mdsderec.h	Destinations master
mdsfacil	mdsfarec.h	Facilities – what special <System_Name> services can be used
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdspasts	mdsstrec.h	Pager stats master
mdsports	mdsporec.h	Ports security
mdsqqb.txt	mdsqarec.h	Bulk queue
mdsqxx.txt	mdsqrec.h	Destination queues
mdssequ	mdssqrec.h	Message sequence number
mdstimev	mdsterec.h	Timed events
== None ==	mdsqdrec.h	Detailed Bulk Queue Data record (Message + data)

Table 119: skymds20: Database Tables

23.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description
MDS_DATA_DIR	The directory where the message queues (and history) exists.
MDS_BIN_DIR	Where the executables are, usually /opt/mds/bin
MDS_MACHINE_ID	What server are we using (M,O = Melbourne, B=Brisbane, ...)

Table 120: skymds20: Environment Variables Us

23.5 IMPORTANT DATA TYPES

Type	Name	Description
Queue Header	QHREC	Next empty/free message + next message to send + how many sent
Details of a message	QDREC	Pager/contact + date/time + booked call? + Billing + where from + lots more ...
Message + Details	NQUREC	QDREC + message + sent/unsent

Table 121: skymds20: Important Data Types

23.6 SKYMD20: MAJOR FUNCTIONS

23.6.1 FUNCTION: MAIN

- call `initialise()` to establish skymds20's run-time environment
- `CDR_start_session()` opens the CDR file
- In an infinite loop

This is the loop that reads each message (using `get_next_mess()`) and uses `queue_all()` to send it to the appropriate destination queue.

- test the shutdown semaphore. Should I kill myself?
- gets a message with `get_next_mess()`.

If no message is available – the bulk queue is empty – skymds20 sleeps for 4 seconds.

- Test whether this message should be ‘skipped’, not posted to any queue.

If this message should be posted (not skipped), `queue_all()` sends it off towards a destination queue

- `update_queue()` updates the state of the bulk queue after each message is read.
- either, go round the loop again, or call `CDR_end_session()` to close the CDR file.

23.6.2 FUNCTION: INITIALISE

- read the environment variables needed
- parse the commandline
- work-out whether ‘dual destinations’ are being used. Whether there are or not is irrelevant as dual queues are no longer used
- call `background()` to put skymds20 into the background, i.e. run as a daemon
- catch and handle a few signals
- check the destination database table (mdsdests) for the current city and queue type (bulk-SMS or normal), and that the port (e.g. /dev/dum07) passed on the command-line matches is correct
- record the port in the mdsports table
- read the details of the shutdown semaphore
- open all the database tables needed

after this is a stretch of code commented out inside an `#ifdef oldver` conditional compilation block.

- open the destinations database table (again)
- call `queue_open_all()` to open all the output queue files, and record their details – including the file descriptors – in the ‘q[]’ array.
- open the timed events file. This is where reminders and booked messages are stored for future, timed, delivery.
- call `mnt_queue_open()` to open the maintenance queue. This is used to keep the data in the various <System_Name> installations’ databases consistent
- use `plock()` to keep skymds20 from being swapped out of memory

23.6.3 FUNCTION: GET_NEXT_MESSAGE

As the name suggests `get_next_message()` reads the next message from the bulk queue

- `QU_queue_read_msg()` reads the queue header and the next ‘message+details’ record into two global variables.
- The ‘nextavail’ (next free record) is compared to ‘nextsend’, the next record we can send. If these two are the same then there are no free records: nextavail is greater than nextsend when there are unsent messages in the queue. In this case the shutdown semaphore is set to ‘1’ to indicate that the queue is empty.
- A check is made in case this is an “old” queue record.

The data records used to be smaller in size, and this is a check that one of these older, smaller records hasn’t accidentally been left in the system.

23.6.4 FUNCTION: QU_QUEUE_READ_MSG

`QU_queue_read_msg()` is coded in mdsqrtn.c, a collection of general purpose queuing functions.

- `QU_queue_read_hdr()` reads the queue header
- `QU_queue_unlock_hdr()` wraps a call to `lockf()` that unlocks the queue
- That there are messages in the queue is now checked. If not, `QU_queue_read_msg()` exits.

The test is actually the same as that done in the calling `get_next_message()` function. It never hurts to be safe.

- Using lseek and then `read()`³⁷, the next record / message is read from the queue.
- `QU_queue_msg_validate()` is now used to check that the message is OK.
- Finally the details, the remainder of the record that isn’t the message, are now read from the queue.

23.6.5 FUNCTION: UPDATE_QUEUE , QU_QUEUE_UPDATE_HDR

`update_queue()`, coded skymds20.c, just calls `QU_queue_update_hdr()` (in mdsqrtn.c) to update the queue header after a message has been read from a queue.

`QU_queue_update_hdr()` executes the following steps:

- calls `QU_queue_read_hdr()` reads the queue header
- the count of the number of messages sent via this queue is increased
- The alternative ‘next_send’ variable is advanced by the message length
- The (changed) header is written back to the queue.
- The queue header is unlocked.

It is not locked during this function. This is probably just to ensure that it is unlocked upon the ending of a message-read cycle.

23.6.6 FUNCTION: QUEUE_ALL

- if this message is from one of 5 test operators, set the info_pager flag to true so that the message isn’t sent
- the message length is calculated
- `queue_all_pagers()` posts the pager message
- if this is meant to be a repeated message, then, as many times as necessary, call `message_to_timed_events()`.

³⁷

NB: not fgets() or the like. Using the unbuffered read function

23.6.7 FUNCTION: QUEUE_ALL_PAGERS

- first check whether messages to this pager should be forwarded to another pager

An “Alternative Pager” contact (advanced processing) service is set up. If there is an alternative pager set up, the record to be posted is updated by `check_substitutions()` to reflect the new pager. Note that the comments at the start of the mdsqall.c source file indicate that this function call has been commented out, but, at the time of writing, it is not.

Most of the remainder of `queue_all_pagers()` concerns grouped pagers, where one pager number represents a collection of ‘real’ pager numbers.

Grouped Pagers:

- Testing flags and calling `scheck_delayed_delivery()`, see if this message should actually be posted another time
- send a pager message to the pager number that identifies the group

This may or may not be a real pager. It could be a pager belonging to a person who wished to have all (or some) of their pager messages duplicated, e.g. the head of a company department. Alternatively it may be an artificial pager number that only represents the group. In this case the message is sent to a null destination, i.e. dumped.

- In a loop over all the pagers in the group:
 - check (via the facilities table) that this is (correctly) a grouped pager
 - check that the data for this pager is valid
 - check that the pager is not inactive
 - copy the details of this pager to the record to be written to the queue
 - `queue_single_pager()` posts the record (message) to the destination queue

Normal pagers:

call `queue_single_pager()` to post the message to the queue.

23.6.8 FUNCTION: QUEUE_SINGLE_PAGER

- Testing flags and calling `scheck_delayed_delivery()`, see if this message should actually be posted another time
- If appropriate, look up the sequence number table (mdsseq) to append an incremental sequence number to the pager message. `increment_sequ_nbr()` is used to increment the pager number
- If a pager’s destination is blank:
 - Convert the destination to NULL
 - Check for “follow me” – destinations in an alternative city
 - Use `check_group_dests()` to check for the group destinations, e.g. if this pager is the member of a group
- If a pager’s destination is not blank:
 - if there is more than one destination, call `check_group_dests()` on each
 - else, presume that it is an OK destination.
- Don’t write a history record if the message has come from skymds27 (contact processing), from the IVR, or flagged to not need a history record
- Write a history record for the rest:

Primary History: `setup_and_write_hist_primary()` calls `HIST_write_primary()` to write the primary history record

Secondary History: `write_msg_hsec_rec()` calls `HIST_write_sec()` to write the secondary history record
SMS: `write_deliv_hsec_rec()` calls `HIST_write_sec()` also to write a secondary history record

Cycle through the destinations for this pager message.

- See if we need to write an alternative CDR record for this pager.

By testing each destination against the destination array (filled by the call to `queue_open_all()` in the skymds20's `initialise()` function) we see if any have their '`alt_route_flag`' set. If they do – indicating that the message may travel to one of many alternative destinations – we will need to set up an alternative CDR record. Alternative CDR records are held in a temporary store for possible updating, before being written to the CDR store proper. Currently only SMS messages, being delivered by one of many SMS carriers, have uncertain destinations, though in the future, well, who knows how messaging may work. Maybe just throw it at the internet.

- Post the message to the queue (you've heard that before, huh?), with `queue_single_dest()`
- If a dual source / destination pair was given on the command line, and the current destination is the 'source' dual destination, then call `queue_single_dest()` again to send the message to the dual 'destination' destination.
- Write the CDR, using either `log_alt_cdr_rec()` or `log_cdr_rec()` for alternative and normal CDRs respectively

23.6.9 FUNCTION: QUEUE_SINGLE_DEST

- check that the queue is open, else call `queue_open_one()` to open it
- copy the message passed into `queue_single_dest()` to an output message
- Do some work on this output copy, get it all ready to go
- Get the file pointer to the destination queue from the array of destinations
- Post the message to the queue (you've heard that before, huh?), with `QU_queue_write_msg()`

23.6.10 FUNCTION: QU_QUEUE_WRITE_MSG

- use `QU_queue_write_hdr()` to update the nextavail (next available) position in the queue.
- increment the semaphore for this queue
- `lseek()` to the appropriate position in the queue
- `write()` the message to the queue. Yes, it's actually happened!
- re-`seek()` to the start of the queue to ensure that it is not corrupt
- lock the queue file

23.7 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

23.8 STOPPING SKYMD20

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)
3. locking the bulk queue would also work, but that's a bit of a hack.

24 mdsalarm

24.1 MDSALARM OVERVIEW

mdsalarm monitors various <System_Name> processes and devices, raising an alarm if it looks like there are problems of any sort.

At the time of writing, mdsalarm will raise an alarm (a pager message with escalations):

- when disk space is below a set minimum
- when any of the message queues have grown too big, holding a backlog of unsent messages, or are not sending messages at all.
- when the database update queues are, likewise, too big or not moving.
- when the encoders don't seem to be working normally
- when failures are detected during checking that the ports in use are actually active and data is being received
- when the doesn't seem to be enough network traffic
- when the history files don't look OK
- when any program has crashed, or any program that should be running isn't.
- when there too many reminders (Call Center Operator 'To Do' tasks.)

24.2 RUNNING MDSALARM

mdsalarm is run, like most <System_Name> daemons, from the <System_Name> start.mds start-up script.

24.2.1 COMMAND LINE PARAMETERS

Parameter	Description	Default
-D	run in debug mode, listing detailed run-time information.	
-n	don't check network delay times	-- no default --
-g	do check the SAGRN (South Aust Govt) CDR files	-- no default --
-r < number >	the maximum number of reminders allowed before an alarm is raised	20 reminders
-t < port >	the port that mdsalarm is registered on	40 messages
-d < num. MB >	the minimum space (MB) that must remain on each disk	70 messages
-q < num. messages >	the maximum allowable size of any queue before an alarm is raised	60000000 messages
-u < num. messages >	the maximum maintenance queue length before an alarm is raised	7 day's worth
-s < num. messages >	the maximum allowable length of the queues.	-- no default --
-h < num. days >	alarm if there aren't enough days of future history files created	
-o < db servers(s) >	the oracle database servers that (this copy of) mdsalarm should monitor	

Table 122: mdsalarm: Command-Line parameters

24.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute mdsalarm at one <System_Name> site is:

```
$MDS_BIN_DIR/mdsalarm -o "SSVIC/SSADE" -q 30 -d 100 -u 70 -t /dev/dum03 2>$LOG_SCR
```

Indicating that mdsalarm is to:

- monitor the SSVIC (Melbourne) and SSADE (Adelaide) Oracle <System_Name> II database servers
- raise an alarm if more than 30 messages in the bulk queue
- raise an alarm if less than 100MB of space is left on any hard disk
- raise an alarm if more than 70 messages in the maintenance queue
- register as running on the port /dev/dum03
- copy messages to stderr to the location LOG_SCR, generally set to /dev/tty0p5 in start.env

24.3 PROGRAM DESCRIPTION

mdsalarm monitors the <System_Name> system(s), raising one or more alarms (pager messages) if problems are discovered. Basically it (mdsalarm's `main()` loop) cycles through each test, followed by a wait for `sleep_time` seconds, `sleep_time` being read from the ports table for the port passed in on the command line. `sleep_time` is currently 300 seconds, i.e. the <System_Name> systems are tested once every 5 minutes.

24.4 PROGRAM STRUCTURE

Following are listings of the components that comprise mdsalarm.

24.4.1 SOURCES FILES

Source File	Description
alarmrtn.c	Generic alarm routine.
form_path.c	Make a file-path from an environment variable and a filename
gensubs.c	Contains various general-purpose routines that are separately compiled.
globvar.c	Functions to access the Global Variable shared memory used by multiple programs.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
mdsalarm.c	System alarming – Check that processes have not aborted.
mdscdr.c	Contains various general-purpose routines that are separately compiled.
mdsemqmain.c	Generic message email queuing routine.
mdsginit.c	A number of functions that perform commonly used initialisation code segments.
mdskbld.c	Contains functions to build and unbuild pager keys, and set-up a dummy pager.
mdslog.c	Generic error logging.
mdsoracle.c	Functions to connect <System_Name> to an Oracle database, as in <System_Name> II.
mdsphol.c	Functions to attach, check and modify Public Holiday shared memory.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurtn.c	Generic message queuing routine.
mdsrem.c	Functions to access the reminders queue, used by multiple programs
mdssemeck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssetup.c	Obtain set-up definitions from a text set-up file.
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions, which perform operations to do with time.
spawn.c	Spawns off a child process and (optionally) waits for its completion.

Table 123: mdsalarm: Source Files

24.4.2 HEADER FILES

Header File	Description
gentypes.h	User defined types and booleans
globvar.h	Global shared memory structures
macros.h	User defined macros.: definitions & macros for common functions
mdsacrec.h	Alternate CDR file
mdsalrec.h	MDSALARM definition file
mdsaprec.h	A-Party Master file
mdscdr.h	CDR definitions file
mdsctype.h	Used for validating message chars.
mdsderec.h	Ports security master file
mdsdlrec.h	Networking Delay monitoring file
mdsemqu.h	Email Queue file
mdsflpsz.h	Standard <System_Name> includes.: field length & definitions
mdsgldr.h	CDR data structure for SARGN (Sth Aust Govt) dealings
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsncrec.h	Network Connection master file
mdsparam.h	parameters used by PMS system

Table 124: mdsalarm: Header Files

Header File	Description
mdsparec.h	Pager Master File
mdsphol.h	Public Holiday Shared Memory file
mdsporec.h	Ports security master file: Ports Record
mdsquarec.h	Contacts Message queue file: Destination Message queue file
mdsrem.h	Reminders shared memory definition
mdsrprec.h	Reminders parameters master file
mdstrrec.h	MNTQ: maintenance queue file

Table 125: mdsalarm: Header Files (continued)

24.4.3 DATABASE TABLES

Each of these header files contains (only) a C-struct that matched the structure of a database table used by mdsalarm.

Database Table	Header File	Description
mdsacrec	mdsacrec.h	Alternative CDR records
mdsaparty	mdsaprec.h	A-Party Customer Info
mdsdestm	mdsderec.h	Destinations master
mdsdelay	mdsdlrec.h	Delay monitoring
mdsline	mdslirec.h	Indial number information
mdsnetcn	mdsncrec.h	Network connection
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsquarec.h	Destination queues
mdsremp	mdsrprec.h	Reminders parameter
mdsqa1.txt	mdstrrec.h	Inter processor transaction queue

Table 126: mdsalarm: Database Tables

24.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Example	Description
MDS_BIN_DIR	/opt/mds/bin	All the <System_Name> executables
MDS_DATA2_DIR	/var/mds/data2	Another data directory - though set may not exist
MDS_DATA_DIR	/var/mds/data	The root of the data directories, queues, database and history.
MDS_MACHINE_ID	O	M,O Melbourne S,X Sydney A Adelaide ...
SS_DB_PASSWORD	s556f08e094tzqfh64...	A <System_Name> II oracle server login password
SS_DB_USERNAME	larry_ellison	A <System_Name> II oracle server login name

Table 127: mdsalarm: Environment Variables

24.5 MDSALARM: MAJOR FUNCTIONS

24.5.1 FUNCTION: MAIN

The central tasks of mdsalarm are all called from its `main()` function which basically consists of a loop that executes each system-testing sub-function, sleeps for `sleep_time` (5 minutes) and goes through the process again.

- `initialise()` is called to prepare mdsalarm
- `check_hist` and `check_network_connection` are both forced to be true if it is before 1:00 in the morning, effectively turning on history file checking and network delay checking regardless of command-line parameters as these values are not set back to their original values.
- `check_processes()` ensures that all the processes that should be running are running
- `check_queues()` tests how many messages are waiting on
- `check_eml_queues()` tests that the emails are going out as they should
- `check_acdr_rec()` ensures that there are not too many (temporary) alternative CDR records
- `check_queue_mq()` checks that the maintenance queue is operational
- `check_ss_bulkq_db()` – are the <System_Name> II oracle message queues working?
- `check_activity()` checks that all the ports that should have applications running on them infact do
- `check_disk()` raises an alarm of there is less than the stipulated amount of space left on any hard disk
- `check_disable_alarm()` raises an alarm if any (other) alarms have been disabeld for too long.
- if the ‘-g’ parameter was passed in on the command-line, the SAGRN CDR record(s) are checked by `check_grncdr_record()`. This will only be done on Inkg, the South Australian Government’s <System_Name> server.
- If this is the first time through the loop, always test the existence of the history files and the length of the network delay with `check_hist_files()` and `check_network_connection_file()`.
- if it is after 9:00am, turn the history file and network connection testing off.
- If this is the master (or only) server at a site then use `check_reminder()` to ensure that there are not too many reminders (to-dos for the Call Centre Operators) backed up.
- If network delay monitoring is turned on, `check_network_delay_file()` checks that the mdsdelay table is, in fact, being written to.
- Finally `sem_check()` is called to test the shutdown semaphore. The `sleep_time` value (300 seconds) is passed to it, telling `sem_check()` to wait 5 minutes before returning. If a non-zero value is returned, then it is time to shut down and the loop is broken.

Now go around again

24.5.2 FUNCTION: INITIALISE

The `initialise()` function sets up mdsalarm’s run-time environment, via the following steps:

- Read in the necessary environment variables
- Process the command line. See ‘Command-line Parameters’ above.
- call `background()` put this mdsalarm process into the background, i.e. to run as a daemon.
- call `closefiles()` to close all open files except stdin, stout and stderr
- catch most common signal by setting `sig_dump()` as their handler.
- call `PORT_set_port_started()` to record in the mdsparts table that the port (/dev/dum03 or whatever) is in use
- read the shutdown semaphore ID and the current ‘operator’ from the port record. The operator will be *mdsalarm*.
- open all the database tables needed.
- now read through all the destination queues. For queue each that sends it messages to a further destination queue (instead of straight to some message delivery device), replace the original queue with the further ‘route via’ queue in the list of queues to test.
- read all the ports (from the mdsparts database table) and record which we need to monitor for lack of activity. Also record the maximum allowable time since the last usage of the port.
- a dummy call to `check_activity()` is now made as the first call to this function always returns true.
- split up the string of <System_Name> II oracle servers passed on the command line to (say) “SSVIC” and “SSADE”
- record the <System_Name> server (e.g. Inkm) we are running on.
- connect to the global and the reminder shared memory blocks
- call `mnt_queue_open()` to open a file handle to the maintenance queue which transfers messages and updates between <System_Name> sites

24.5.3 FUNCTION: ALARM_PROCESS_ALARM

ALARM_process_alarm(), coded in alarmrtn.c, is called by each of the monitoring functions (coded in mdsalarm.c) to initiate the raising of the alarm. After the logic of the monitoring function determines that an alarm-worthy condition does, in-fact, exist, control is passed to **ALARM_process_alarm()** to create and send the alarm.

- collect the current time and operator
- open the alarm database table, mdsalarm
- look for the alarm type in the alarm table. If the alarm type is *not* found, **ALARM_send_message_local()** is simply called to send an alarm pager message to a local support pager.
- If the alarm has been disabled then exit. Many alarms are disabled overnight, though some will be disabled for other reasons.
- If another alarm has been sent recently, more recently than the number of seconds stipulated in this alarm type's alarm_freq database table field, **ALARM_process_alarm()** will also be exited.
- either **ALARM_send_message_local()** or **ALARM_send_message_remote()** are called to send the alarm to either local or remote <System_Name> support personnel.
- Now that the alarm has been sent, the 'last alarm time' and the last operator to raise this alarm are recorded.
- finally the other <System_Name> servers' mdsalarm tables are updated via the maintenance queue

24.5.4 FUNCTION: ALARM_SEND_MESSAGE_LOCAL

Send an alarm to a local support person.

- call **ALARM_validate_pager()** to ensure that the pager number to which the alarm is being sent actually exists
- **ALARM_setup_queue_rec()** prepares the bulk queue <System_Name> message (i.e. the alarm)
- **queue_main()** posts the message (the alarm) to the bulk queue.

24.5.5 FUNCTION: ALARM_SEND_MESSAGE_REMOTE

Send a message to a remote server. Basically this function is a simple, short wrapper to execute the 'page' executable on some non-local <System_Name> server. It works like this:

- a command line to run **remsh** is constructed, remsh being a Unix operating system command that runs processes on foreign, accessible servers. In this case remsh is running the rpage script on some non-local <System_Name> server
- a **system()** call runs the **remsh** command, **remsh** runs **rpage**, and **rpage** runs **page**.
- page sends a pager message (the alarm) to the appropriate pager

24.5.6 FUNCTION: CHECK_PROCESSES

check_processes() reads through the mdsparts database table, checking that each processes registered against a port has either recorded an end time (i.e. it has completed normally) or is still running.

For each process recorded in the mdsparts database table:

- is an end time recorded? If it is, then this process is all OK
- test the recorded pid of the process to see if it is still running
- if the process is not running:
 - write a count into the end-time field of the message indicating how many times this alarm has been raised
 - call **ALARM_process_alarm()** to raise an alarm, passing the flag ALARM_PROCESS, or ALARM_BULKQ if the process is skymds20³⁸.

³⁸ skymds20 extracts messages from the bulk queue and inserts them into the distribution queues

24.5.7 FUNCTION: CHECK_QUEUES

`check_queues()`, as the name suggests, checks each of the queues to ensure that too many messages haven't built up in them. The maximum number of messages allowed is either passed in on the command line or the (impossibly vast) default is used.

In a loop, over each queue:

- determines how many messages are in the queue by reading the queue header
- compares this number against `max_qu_filesize`, set by either the `-s` command line parameter or against the default of 60,000,000. The default represents about two years of messages. If the queue exceeds the maximum allowable an alarm is raised with `ALARM_process_alarm()`
- Now a similar test is done, but rather than test the length of the queue it compares the number of messages *entered* with the number actually *sent*, this difference being compared against (a copy of) the `alarm_threshold` value for the queue in the `mdsdests` table. `ALARM_process_alarm()` is again used to raise an alarm if the number of unsent messages is too high.

24.5.8 FUNCTION: CHECK_EML_QUEUES

Check the state of the email, email to SMS and (email to) Bulk SMS queues.

- set the limit for the size of the queues to `q_max` (the '`-q`' command-line parameter), and $2 \times q_{max}$ for the Bulk SMS queue
- go through all the email queues, adding up the unsent message counts by queue type; email-to-pager, email-to-SMS, and email-to-BulkSMS.
- now go through the collection of unsent message counts. If any type of queue(s) has (have) collected more than their limit of messages then use `ALARM_process_alarm()` to raise an appropriate alarm.

24.5.9 FUNCTION: CHECK_ACDR_REC

Alternative CDRs are billing records (CDRs) that will be updated with further information after creation. As it is time consuming (and possibly corrupting) to locate and extract CDRs already stored, alternative CDRs are made and retained separately for updating.

Currently (Oct. 2002), the only update made to CDRs is the carrier that transmits SMS messages to their destination. Thus all SMS messages write an alternative CDR that, a few minutes after it's created, will be rewritten to a proper CDR. `check_acdr_rec()` ensures that there are not too many of these alternative CDRs waiting to be updated.

- `isindexinfo()` is used to get the number of alternative CDR records in the `mdsacrec` (alternative CDR) table.
- if this number is over 200, or nothing has moved for 30 minutes then construct an alarm message
- if it is between 9 in the morning and 8 in the evening raise an alarm, otherwise discard. The situation will either correct itself (somehow), or it can be dealt with tomorrow morning.

24.5.10 FUNCTION: CHECK_QUEUE_MQ

`check_queue_mq()` checks the state of the maintenance queues, the queues that are used in the transportation of messages and database updates between the various <System_Name> servers.

- first the size of the maintenance queue is read from the queue's header
- if the queue seems to be halted, then use `ALARM_process_alarm()` to send a message.
- if the queue has more messages than the amount passed into `mdsalarm` by the '`-u`' command-line parameter, or than the default limit (70) if the `-u` option wasn't used, then an alarm is likewise raised.
- this second test is then repeated, comparing the physical size of the queue with the maximum allowable messages x the size of each message. I don't know what the reason for this second test is.

24.5.11 FUNCTION: CHECK_SS_BULKQ_DB

check_ss_bulkq() tests the <System_Name> II message queues on one or more <System_Name> II Oracle servers. The list of <System_Name> II servers is passed in on the command line, e.g. “-o SSVIC/SSADE”.

The bulkq in the name refers to the <System_Name> II message queue, which is also called the bulk queue.

Looping over each <System_Name> II oracle database servers, **check_ss_bulkq_db()**:

- attempts to connect to the server. If this fails an alarm is raised.
- if there is greater than “q_max” (-q command line parameter) messages in the <System_Name> II message queue, or the oldest message has been in the queue more than 10 minutes, then raise an alarm.

24.5.12 FUNCTION: CHECK_ACTIVITY

Check all the ports on which we would expect there to be activity. These were selected during the **initialise()** function and stored in the **alarm_port[]** array.

In a loop over all of these ports:

- some ports are not to be checked during the weekend. If it is the weekend then skip these.
- if the last-use time in the port’s shared memory location is blank, then insert a time so that we can test it next time.
- if the time since the last usage of this port is greater than that permitted as per the **alarm_ports[]** array then call **ALARM_process_alarm()** to raise an alarm.

24.5.13 FUNCTION: CHECK_DISK

Check disk uses the **statvfs()** function to ascertain the space available on the server’s hard drive and compares this to the minimum allowable space.

- use the **statvfs()** function to gather a whole lot of information about the hard-drive.
- extract the amount of space available on the Hard drive from the information returned
- if this is less than the minimum permissible, as determined by the ‘-d’ command line parameter, **ALARM_process_alarm()** is called to raise an alarm.

24.5.14 FUNCTION: CHECK_DISABLE_ALARM

During working hours, this function checks each disabled alarm (**disable_alarm =='Y'**) to see if it has been disabled for more than a day. If one has and alarm is raised to this effect (i.e. an alarm has been disabled for too long).

Many alarms are disabled during the day as, while important, they can wait till tomorrow to be fixed.

24.5.15 FUNCTION: CHECK_GRNCDR_RECORD

The South Australian Government runs their own <System_Name> server, lnkg, to service the SAGRN: the South Australian Government Radio Network. On this server (only) we must check that the South Australian Government’s CDR’s are being written as an indicator that their (and our) network(s) is/are functioning properly. The logic of this function is to test that at least one (South Australian Government) CDR has been written during the last hour. “grin” is the ‘code’ for “Government Radio Network”

- open the SAGRN CDR file.

This file is called GRNCDRccyyymmdd (e.g. GRNCDR20031104) and lives in the /var/mds/data2/ directory, which itself will only be found on the lnkg machine.

- the time since midnight and the time since the last CDR update are ascertained. If both of these are greater than an hour and **ALARM_process_alarm()** raises an alarm.

24.5.16 FUNCTION: CHECK_HIST_FILES

`check_hist_files()` ensures that there are always 7 (or whatever) empty history files waiting to be filled.

- add the number of days stipulated in the **`globvar.txt`** file + either the number passed on the command-line as the ‘-h’ parameter or the default (7) + 1. Generally this will mean that 7 +7+1=15 days’ worth of files must be created or an alarm is raised.
- make the appropriate filename from the current date. History files are named by date, e.g. (the pair) **`hi20020730.dat`** and **`hi20020730.idx`** (primary history) and **`hs20020730.txt`** (secondary history).
- check that the primary and secondary files with the appropriate names exist. If not, call **`ALARM_process_alarm()`** to raise an alarm.
- Increase the date and go round again.

This is done ‘how-many-days’ times, e.g. 15.

24.5.17 FUNCTION: CHECK_NETWORK_CONNECTION_FILE

This function tests whether any messages are being delivered via ports other than their defaults. If any are it indicates a problem.

- open the **`mdsnetcn`** (network connection) and the **`mdsdestm`** (destinations master) database tables
- each destination-to-encoder (“ENC”) record is read from the network connection file
- Next we find the destinations master record with destination queue mnemonic matching that in the current destination-to-encoder record.
- from that destinations master record we find the port it should be communicating via, and then the matching record for that port in the ports table (**`mdsports`**).
- finally the **`conn_addr`** field in the chosen record in the **`mdsports`** table is compared to the **`host_name`** and **`host_addr`** fields in the current network connection table. If these are not the same then **`ALARM_process_alarm()`** raises an alarm

This test is done for each destination-to-encoder (“ENC”) record in the network connection (**`mdsnetcn`**) table.

24.5.18 FUNCTION: CHECK_REMINDER

`check_reminder()` ensures that there are not too many reminders outstanding. The user can set the maximum number of reminders by using the ‘-r’ command-line parameter, or the default maximum of 20 can be used.

- open the reminders table

The reminders themselves are kept in shared memory. What happens is that two fields in the reminders database table (**`node_index`** and **`queue_index`**) provide indexes into the shared memory block. At the location identified a queue header is read which contains the number of reminders in this particular queue. It is this number that is tested.

- generate an index into the reminders shared memory and test the number of reminders. If this number is larger than the maximum allowed, call **`ALARM_process_alarm()`** to raise an alarm
- Loop onto the next reminder table record. This will provide two new indexes into the reminder shared memory

24.5.19 FUNCTION: CHECK_NETWORK_DELAY_FILE

`check_network_delay_file()` only ensures that the **`mdsdelay`** database table is being written to. It doesn’t test the length of the delays itself. The network delay table records how long it is taking to send a message in each of the six <Company_Name> frequencies.

This test is suppressed if **`mdsalarm`** is run with the ‘-n’ option.

- get the current time
- find the latest network delay record
- convert the time lapse between now and when this last delay record was written
- test the time lag against **`NBR_MIN`**, which is #define’d to be 25 (minutes)

As before, if the time limit has been exceeded, call **`ALARM_process_alarm()`** to raise an alarm.

24.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

24.7 STOPPING MDSALARM

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)

25 skymds23

25.1 SKYMD23 OVERVIEW

skymds23 reads <System_Name> messages from the local maintenance queue, passes them through the <System_Name> network where they are read by one or more non-local instances of skymds24. skymds24 writes these messages to the (non-local) <System_Name> system. This system (skymds23, skymds24) is used to keep the databases at each <System_Name> site synchronised with each other. All database updates are replicated in this way, so that *each* <System_Name> site contains all the data necessary to run *all* the other <System_Name> sites. The diagram on the next page provides a representation of inter-city updates via skymds23 and skymds24.

Note that there are a few versions of skymds23, all very similar. skymds23i is currently used, but I will just refer to it as skymds23.

25.2 RUNNING SKYMD23

When running the <System_Name> start-up script start.mds, the operator / user / supervisor is questioned thus:

Start skymds23i and skymds24i (y/n) :

presuming that they answer ‘y’, skymds23 is run.

25.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-a <??>	Two characters, representing city and machine skymds23 is running on / in (e.g. SX)
-b <??>	Same, representing target city and centre (e.g. MO)
-t < comms port >	The communications port skymds23 is registered against (in mdsports)
-h <name>	The name of the machine skymds23 is running on, e.g. lnks
-D	run in debug mode, listing detailed run-time information.
-d	Print out a message for each packet write (Display Packet)
-p <number = address >	The hardware port address through which socket / network communication travels
-s	hist_skip_mode: don't transmit history updates
-n	sequ_skip_mode: don't inter-site update message sequence number
-r	remq_sequ_mode: don't inter-site update reminders
-j	timev_skip_mode: don't update timed events
-v	don't send any messages that are not directly targeted to the destination. No route <u>Via</u> print (incomplete, out-of-date) usage instructions to stdout.
-?	

Table 128: skymds23: Command-Line Parameters

25.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute skymds23 at one <System_Name> site is:

```
$MDS_BIN_DIR/skymds23i -a SX -b SS -v -p 2003 -h lnks -t /dev/dum08 2>$LOG_SCR
```

indicating that skymds23 is run:

- on machine lnkx (X) in Sydney (S)
- sending messages to the Sydney (S) machine in Sydney (S)
- sending any messages directly destined for the Sydney machine only.
- using the hardware (port number) 2003 to communicate to the network
- the machine we are running on is lnks, though this conflicts with the -a option.
- recorded against the port /dev/dum08 in the mdsports table.
- stderr (file number 2) is redirected to the device LOG_SCR, defined in start.env to be /dev/ttys1p0, a terminal somewhere.

25.3 PROGRAM DESCRIPTION

skymds23 is a communication program, sending <System_Name> system messages from the maintenance queue to a skymds24 application at another site. Running as a daemon it extracts the next message from the maintenance queue (`get_next_mess()`), tests various ‘should I send’ conditions, before sending it across the LAN (via sockets) to a skymds24l running on another site.

On a site with multiple skymds23 destinations, there is only one maintenance queue, but with multiple headers. When a message is ‘deleted’ from the queue, all that actually happens is that the `next_message` pointer in the appropriate queue header is advanced, which means that the message is still available to other ‘queues’ to process.

25.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds23.

25.4.1 SOURCES FILES

Source File	Description
gensubs.c	Contains various general purpose routines that are separately compiled.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
mdsginit.c	A number of functions, which perform commonly, used initialisation code segments.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the ports file.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdstmrtn.c	Contains functions that perform operations to do with time.
skymds23l.c	Extract messages from the maintenance queue (mdsqa1.txt) and send them via the network to skymds24 on another machine where they will update files.

Table 129: skymds23: Source Files

25.4.2 HEADER FILES

Header File	Description
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.: C time field definitions
macros.h	User defined macros.: definitions & macros for common functions
mdsderec.h	Ports security master file
mdsflsz.h	Standard <System_Name> includes.: field length & definitions
mdsparam.h	parameters used by PMS system
mdsporec.h	Ports security master file: Ports Record
mdstrrec.h	MNTQ: maintenance queue file

Table 130: skymds23: Header Files

25.4.3 DATABASE TABLES

Each of these header files contain (only) a C-struct that matched the structure of a database table used by skymds23.

Database Table	Header File	Description
mdsdestm	mdsderec.h	Destinations master
mdsport	mdsporec.h	Ports security
mdsqa1.txt	mdstrrec.h	Inter processor transaction queue

Table 131: skymds23: Database Tables

25.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description	Example
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide ...	O
MDS_DATA_DIR	The root of the data directories, queues, database and history.	/var/mds/data

Table 132: skymds23: Environment Variables

25.5 SKYMD23: MAJOR FUNCTIONS

25.5.1 FUNCTION: MAIN

- call **initialise()** to prepare skymds23's run-time environment

now start the infinite loop, reading and sending messages

- first test the shutdown semaphore, breaking out of the loop (terminating the program) if it has been changed to zero.
- next test the queue semaphore. Sleep if
- call **get_next_mess()** to get the next message from the maintenance queue
- Now test a number of conditions under which this message should not be sent.

Do not deliver the message if any of the following (pairs of) conditions are true:

Condition 1	... and ...	Condition 2
reminder message		(-r command-line parameter) ignore reminders
timed event message		(-j command-line parameter) ignore timed events
history update message		(-s command-line parameter) ignore history updates
sequence number message		(-n command-line parameter) ignore sequence numbers
this machine is not the destination machine		(-v command-line parameter) 'via' mode not set

If the destination and source machines of this maintenance queue message are the same.

Table 133: skymds23: When to Not process a maintenance queue message

- if the message should be delivered then call **mess_to_lan()** to deliver it
- whether or not the message was sent, call **update_queue()** to update the next-message pointer in the queue header.

Once the infinite loop is broken, call **finalise()** to close the open database tables.

25.5.2 FUNCTION: INITIALISE

The **initialise()** function sets up skymds23's run-time environment, via the following steps:

- read in the MDS_DATA_DIR and MDS_MACHINE_ID environment variables.
- parse the command-line. See the command-line parameters table above.
- use **background()** to put the process into the background.
- catch a few signals
- use **PORT_set_port_started()** to record in the mdsparts table that skymds23 has started – and what port it is registered against.
- obtain and test the shutdown semaphore
- get the sleep time for the function
- use **mnt_queue_open_for_read()** to open (a file header for) the maintenance queue.
- call **connect_to_lan()** to open the file (socket) representing the <Company_Name> to the foreign <System_Name> site

25.5.3 FUNCTION: GET_NEXT_MESS

get_next_mess() reads the next message (for skymds23 to send) from the local maintenance queue.

- lock the maintenance queue.
- the queue header is read, checked for validity, after which the **nextavail** and **nextsend** indices are read from it.

These indicate where in the queue the next ready-to-go messages are. Note that there may be 4 (or more) queue headers attached to one queue. Each header has and adjusts its own next-message indices.

- unlock the maintenance queue.
- the (standard) maintenance queue message at the **nextsend** location is read
- within the maintenance queue message there is a record length (rec_len) field, indicating the length of the remainder of the maintenance queue message. This amount of data – the rest of the message – is now read from the maintenance queue.
- A couple of safety checks are performed:
 - is this message ‘unsent’? If not: the queue is corrupt.
 - does it end with an end-of-line character. If not: the message is corrupt.

25.5.4 FUNCTION: MESS_TO_LAN

This function writes the message collected from the maintenance queue (by **get_next_mess()**) to the <System_Name> LAN for transmission to another <System_Name> site.

- first write out the message and a few other details if either in debug mode or the ‘display packet’ flag is set (-d passed on the command line)
- the message is written to the LAN, as a file representing a socket connection to another <System_Name> site. This communication takes place via the physical port as passed into skymds23 with the -p command-line parameter.
- wait 15 seconds for a reply

If no reply in 15 seconds, the connection to the LAN is closed and **connect_to_lan()** is called to reconnect again.

- if the reply is ACSII (octal)25 (Negative acknowledge: transaction bad), then return to the start of this function (**mess_to_lan()**) to try and send the message again
- if the reply is ACSII (octal)30 (transaction skip), then we’re done: the LAN / non-local <System_Name> site didn’t want this message anyway
- if the reply is any other response than ASCII (octal) (Acknowledge: OK) then:
 - close the connection to the LAN
 - call **connect_to_lan()** to reconnect to the said LAN
 - try to send the message again.

25.5.5 FUNCTION: MNT_QUEUE_OPEN_FOR_READ

Opens and checks the state of the maintenance queue.

- open the ports master file (mdsports) and the destinations master file (mdsdests).
- the maintenance queue details are collected from the destination table
- the maintenance queue file is opened.

It may already be open and in use by another program. Here we just open another header to it.

- lock the file, read and test the header, and unlock the file.
- check that the computer that skymds23 is running on is listed in the maintenance file’s queue header.
- read the record for the port from the mdspports table. From this record get the semaphore for the maintenance queue.

25.5.6 FUNCTION: UPDATE_QUEUE

update_queue() changes the queue header after a maintenance queue message has been read.

- call **mnt_queue_lock()** to lock the maintenance queue
- read and test the queue header. Abort if the header is corrupt.
- add the length of the read message to the nextsend index of the header
- write the whole header back to the queue
- call **mnt_queue_unlock()** to unlock the maintenance queue

25.5.7 FUNCTION: CONNECT_TO_LAN

This is the function that actually does the connection to the <System_Name> LAN. It uses sockets for arbitrary destination handling.

- first the name of the machine is obtained
- The configuration records for the sending and receiving sockets are set:

local	remote address	description
sin_family	AF_INET	AF_INET
sin_port	0	2003
sin_addr	0	local machine IP
sin_zero	not set	not set

Table 134: skymds23: socket configuration

- `socket()` is called to create a socket end-point, of type AF_INTERNET (internet) and using the standard SOCK_STREAM type.
- `bind()` is used to connect the socket end point to the address in the send socket configuration record.
- `connect()` then connects this local socket to a remote socket using the remote address configuration data
- If this fails then sleep for 10 seconds and go around again, else all done.

25.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

25.7 STOPPING SKYMD23

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)

26 skymds24

26.1 SKYMD24 OVERVIEW

skymds24 receives messages and updates from non-local <System_Name> servers and updates the local database system with this new information. skymds24 reads these messages through a socket over the <System_Name> LAN, and is the sister program of skymds23, skymds23 sending the messages and updates that skymds24 receives and processes.

Note that there are a few versions of skymds24, all very similar. skymds24l is currently used, but I will just refer to it as skymds24.

26.2 RUNNING SKYMD24

When running the <System_Name> start-up script start.mds, the operator / user / supervisor is questioned thus:

Start skymds23l and skymds24l (y/n) :

presuming that they answer ‘y’, skymds24 is run.

26.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-t < port >	the port that this instance of skymds24 will be registered against
-D	run in debug mode, listing detailed run-time information.
-r	re-Queue mode: after reading the message writes it back to the maintenance queue again.
-s	skip log history mode: do not accept any history data updates
-p < port >	the physical port on which skymds24 listens for data (via a socket connection)
-c < subscriber type >	only process these subscriber types. (A MDS_DATA_DIR/mdsinpager.txt file must exist.)
-?	print usage instructions: similar to this table.

Table 135: skymds24: Command-line parameters

26.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute skymds24 at one <System_Name> site is:

\$MDS_BIN_DIR/skymds24l -p 2002 -r -t /dev/dum8 2>\$LOG_SCR

indicating that skymds24 is being run:

- listening on port 2002 for arriving messages
- in re-Queue mode
- registered against port /dev/dum8 in mdspports
- with stderr (file descriptor 2) copied to the device LOG_SCR, defined in start.env to be /dev/tty1p0

26.3 PROGRAM DESCRIPTION

As described, skymds24 collects updates from the LAN and manages the implementation of these updates to the local <System_Name> database. The **main()** function passes control to **process_inter_machine_update()** to implement the infinite ‘read message → implement changes’ loop. **process_inter_machine_update()** calls **check_for_work()** to read the next maintenance queue message and then **process_transaction()** to implement the database changes indicated or to queue the messages contained in the maintenance queue message.

26.3.1.1 Getting the updates

Check_for_work() calls

- **get_next_message()** to read the message from the maintenance queue
- **validate_transaction()** to ensure that the message / update is not corrupt. It also generates the flag ‘file_id’

26.3.1.2 Implementing updates

`process_transaction()`, a huge switchboard function, processes the update messages obtained by `check_for_work()`. It selects one of 84 or so function calls depending on the value of ‘file_id’ (determined in `validate_transaction()`), each of these functions making an update to one or more tables in the <System_Name> database system. The calls (file_id || function call) are detailed in the `process_transaction()` section below. As clear from that table, `process_rec()` accounts for around 75% of the update function call options, specialised function calls comprising the remaining 25%.

26.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds24.

26.4.1 SOURCE FILES

Source File	Description
skymds24l.c	Get messages from the network (from skymds23l) and use these transactions to update the files.
alarmrtn.c	Generic alarm routine.
gensubs.c	Contains various general-purpose routines that are separately compiled.
globvar.c	Functions to access the Global Variable shared memory used by multiple programs.
isstart.c	Wrapper for C-ISAM isstart function – choose index and position the read-point
mdscdr.c	Contains various general-purpose routines that are separately compiled.
mdsemqmain.c	Generic message email queuing routine.
mdsginit.c	A number of functions that perform commonly used initialisation code segments.
mdshisio.c	Generic history file handling routines.
mdskbld.c	Contains functions to build and unbuild pager keys, and set-up a dummy pager.
mdslog.c	Generic error logging.
mdspolc.c	Functions to attach, check and modify Public Holiday shared memory. Used by multiple programs.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurtn.c	Generic message queuing routine.
mdsrem.c	Functions to access the reminders queue, used by multiple programs
mdssemcck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdstmrtn.c	Contains functions that perform operations to do with time.
mdstxrtn.c	A number of functions, which perform commonly, used initialisation code segments.
security.c	Various Security Routines.

Table 136: skymds24: Source Files

26.4.2 HEADER FILES

Header File	Description
mdsalrec.h	MDSALARM definition file
mdsaprec.h	A-Party Master file
mdsaurec.h	Audit trail master file
mdsbtre.h	Bulletin master file
mdscdr.h	CDR definitions file
mdscdrec.h	CDR data structure
mdsckrec.h	CDR data structure
mdsclrec.h	Client master file
mdsemrec.h	Company Details file
mdscorec.h	(CLT/INV/GRP) Contacts Master File
mdsepcrec.h	Call Patching Master File
mdscsrec.h	Client stats master file
mdsctrec.h	Extended Canned Text File
mdscurec.h	Customer Program File
mdsderec.h	Ports security master file
mdsdlrec.h	Networking Delay monitoring file
mdsdsrec.h	Delivery Status file

Table 137: skymds24: Header Files

Header File	Description
mdsecrec.h	External Contact master file
mdsemblk.h	Block Email file
mdsemrec.h	Email File (Online & Batch)
mdserrec.h	GL3000 definition file: GL3000 error file
mdsesrec.h	(ESC) Escalations Master File
mdsetrec.h	Escalation transaction master file
mdsfarec.h	Pager facilities master file
mdsfldsz.h	Standard <System_Name> includes.: field length & definitions
mdsfmrec.h	(FRM) Forms Master File
mdsftrc.h	Forms text File
mdsfurec.h	System Functions file
mdsfxrec.h	(FAX) Fax Master File
mdsg1rec.h	GL3000 tnpp record: <System_Name> -> GL3000 CDR record layout
mdsg2rec.h	GL3000 Billing record: GL3000 -> <System_Name> CDR record layout
mdsgl3000.h	GL3000 definition file: GL3000 layout
mdshirec.h	History primary recs file: New History logging file
mdshsrec.h	Secondary History file (audits)
mdsiprec.h	Info Pack master file
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsncrec.h	Network Connection master file
mdsnrrec.h	New Rosters Master File
mdsparec.h	parameters used by PMS system
mdsphol.h	Pager Master File
mdsphec.h	Public Holiday Shared Memory file
mdsporec.h	(TMS) Telephone Message Service
mdsporec.h	Ports security master file: Ports Record
mdspsrc.h	Password master file
mdspurec.h	Public Holiday master file
mdsqrec.h	Contacts Message queue file: Destination Message queue file
mdsrem.h	Reminders shared memory definition
mdsrrec.h	Reminders master file
mdsrprec.h	Reminders parameters master file
mdsscerc.h	Service Code Master File
mdssdrec.h	Search Data Master File
mdsserec.h	Security Group master file
mdssfrec.h	Software Request Form file
mdssporec.h	Search Parameter master file
mdssqrec.h	Message sequence number file
mdsssrec.h	SMS stats file
mdsstrec.h	Pager stats master file
mdsterec.h	Timed event file (incl. booked call)
mdstmrec.h	Temporary message master file
mdstrrec.h	MNTQ: maintenance queue file
mdstxrec.h	(TXT) Delivery Stats Master File
mdsunrec.h	Unique Number File
saf/saflfdsz.h	field length & definitions used by SAF
saf/saflgrec.h	Safety <Company_Name> log file
saf/safprrec.h	Safety <Company_Name> Profile
saf/safsprec.h	Safety <Company_Name> Service file
security.h	Structure and defines for security groups
sel/seladrec.h	SEL Advisory Information file
sel/selbrrec.h	SEL Brand Code file
sel/selcorec.h	SEL Controller Type file
sel/selfldsz.h	SEL data field size definitions
sel/selprrec.h	SEL Problem Code file
sel/selserec.h	SEL Serial number file
startel/staa1rec.h	STARTEL Location Maintenance File
startel/staa2rec.h	STARTEL Dealer Maintenance File
startel/stafldsz.h	Startel directory: definition file for STARTEL.
vip/vipa1rec.h	VIP Suburb/Trade/Service File: VIP Database File
vip/vipa2rec.h	VIP Suburb/Postcode Master File

Table 138: skymds24: Header Files (Continued)

26.4.3 DATABASE TABLES

Database Table	Header File	Description
hiccyymmdd	mdshirec.h	the 'logical' (primary) history table
hsyymmdd	mdshsrec.h	Secondary history storage
misadvise	mdsadrec.h	Advisory
mdsaparty	mdsaprec.h	A-Party Customer Info
mdsaudit	mdsaurec.h	Audit trail
mdsbull	mdsbtrec.h	Operator bulletin
mdscallp	mdscprec.h	Call patching master
mdscdrec	mdscdrec.h	Call details master
mdschkpnt	mdschkrec.h	Generic checkpoint
mdsclien	mdsclrec.h	Client master
mdsclsts	mdscsrec.h	Client stats master
mdsemddet	mdsemrec.h	Company details
mdsconta	mdscorec.h	Contacts master
mdscstext	mdscstrec.h	Escalation large canned text
mdscustp	mdscurec.h	Customer program
mdsdelay	mdsdlrec.h	Delay monitoring
mdsdestm	mdsderec.h	Destinations master
mdsdsrec	mdsdsrec.h	Delivery status master
mdsemail	mdsemrec.h	e-mail master
mdserror	mdserrec.h	generic error
mdsescal	mdsesrec.h	Escalation master
mdsestrn	mdsetrec.h	Escalation transaction master
mdsexco	mdsecrec.h	External contact master
mdsfacil	mdsfarec.h	Facilities – what special <System_Name> services can be used
mdsfax	mdsfxrec.h	Fax master
mdsfmtxt	mdsfstrec.h	Holds curses (text) screens as text records
mdsforms	mdsfmrec.h	Forms
mdsfunct	mdsfurec.h	Function codes
mdsgl01	mdsgl1rec.h	GL3000 (Sth Australian Government) transaction protocol (tnpp) master table
mdsgl02	mdsg2rec.h	gl3000 ver 6.00a billing protocol
mdsinpak	mdsiprec.h	Info pack .
mdsline	mdslinec.h	Indial number information
mdsnetcn	mdsncrec.h	Network connection
mdsnrost	mdsnrrec.h	Roster master
mdspager	mdsparec.h	Pager master
mdspassw	mdsprec.h	Password username
mdspasts	mdsstrec.h	Pager stats master
mdsphone	mdsphec.h	Phone master
mdsports	mdsporec.h	Ports security
mdspubh	mdspurec.h	Public holiday
mdsqal1.txt	mdstrrec.h	Inter processor transaction queue
mdsqxx.txt	mdsquarec.h	Destination queues
mdsrem	mdsrrec.h	Reminders queue
mdsremp	mdsrprec.h	Reminders parameter
mdsschd	mdssdrec.h	Search data
mdsschp	mdssprec.h	Search parameter
mdsscde	mdsscerc.h	Service code
mdssecur	mdsserec.h	Security codes
mdssequ	mdssqrec.h	Message sequence number
mdssmsta	mdsssrec.h	Sms stats
mdssrf	mdssfrec.h	Software request form
mdstext	mdstxrec.h	Text header
mdstimev	mdsterec.h	Timed events
mdstmesg	mdstmrec.h	Temporary message master
mdsuniqn	mdsunrec.h	Unique number
safgl	saf/safglrec.h	Safety <Company_Name> log file (table)
safpr	saf/safprrec.h	Safety <Company_Name> Profile
safsp	saf/safsprec.h	Safety <Company_Name> Service file (table)

Table 139: skymds24: Database Tables

26.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description	Example
MDS_DATA_DIR	The root of the data directories, queues, database and history.	/var/mds/data
MDS_BIN_DIR	All the <System_Name> executables	/opt/mds/bin
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide B Brisbane P Perth ...	O
MDS_TEXT_DIR	Message storage directory	/var/mds/data/text

Table 140: skymds24: Environment Variables

26.5 SKYMD24: MAJOR FUNCTIONS

26.5.1 FUNCTION: MAIN

`main()` is a simple function. All it does is:

- call `initialise()` to prepare skymds24's environment
- call `process_inter_machine_update()`, that, in an infinite loop, reads messages from the LAN and implements the appropriate update based on that message
- once `process_inter_machine_update()` is terminated (by the shutdown semaphore) or crashes out – due to a corrupted LAN or whatever – `finalise()` is used to close all the database tables, and skymds24 exits.

26.5.2 FUNCTION: INITIALISE

The initialise() function sets up skymds24's run-time environment, via the following steps:

- read in the needed environment variables, as per the table above
- parse the command-line
- call `background()` to run skymds24 as a background daemon
- Redirect 25 or so signals to the function `sig_dump()`
- prepare the shutdown semaphore. This is read to determine if it is time to stop processing
- open 24 database tables
- construct an array of record lengths for these tables
- `connect_to_lan()` is used to connect to the <System_Name> LAN
- connect to shared memory segments: reminders, public holiday, security and global shared memory

26.5.3 FUNCTION: CONNECT_TO_LAN

`connect_to_lan()` sets up a socket-to-socket connection over the LAN to a non-local <System_Name> installation. The socket type is set to SOCK_STREAM, making the socket appear as a full duplex pipe.

- `socket()` and `bind()` system calls are used to create and open the local end of a socket-to-socket connection, with the port details of the socket to connect to added in
- `listen()` listens for any packets already present over the socket-socket <Company_Name>.
- `accept()` does the same, but waits until a packet is received. Once it is it returns a new file descriptor for the socket <Company_Name>. This is now the socket.

26.5.4 FUNCTION: PROCESS_INTER_MACHINE_UPDATE

`process_inter_machine_update()`, running in an infinite loop, reads messages from the LAN before passing them to `process_transaction()` to update the local <System_Name> installation.

In an infinite loop:

- check the shutdown semaphore. If has been changed to zero, then shutdown.
- call `check_for_work()` to retrieve the latest inter-site update from the LAN
- if we are in re-queue mode (-r was passed on the commandline) then use `mnt_queue_re_write()` to re-write the message back to the maintenance queue
- if we (the local machine) is not the recipient (as detailed in the message's 'dest_centre' field), then we skip this message
- call `process_transaction()` to implement the changes specified in the message.

26.5.5 FUNCTION: CHECK_FOR_WORK

`check_for_work()` gets and checks the next message from the LAN.

- first it uses `get_next_mess()` to obtain the next message
- then use `validate_transaction()` to check the integrity of the obtained message
- if the record is bad, call `send_reply_status()` each time with a TRANSACTION_BAD flag. Once the maximum number of retries has been reached, call `send_reply_status()` one more time with the TRANSACTION_SKIP flag, telling the sender (probably a skymds23 program on another <System_Name> installation) to dump this record and continue. Then sleep for 10 seconds and wait for the next message
- if the record is good use `send_reply_status()` with the TRANSACTION_OK flag, indicating to the sender that all is hunky-dory over here.

26.5.6 FUNCTION: GET_NEXT_MESS

`get_next_mess()` is really just a wrapper for a read from the socket created by `connect_to_lan()`. `get_next_mess()` reads continuously until a properly configured (terminates with an End-Of-Line character) message is read – hopefully the first message read.

- use the `read()` system function to read up to BUFSIZE bytes from the socket
- if there is a problem, close the socket and call `connect_to_lan()` to re-connect to the LAN
- test if the last character is an End-Of-Line. If it is then we are done. If not then go around again.

26.5.7 FUNCTION: VALIDATE_TRANSACTION

Check the:

1. file_id (what this message is about),
 2. transaction type (add, update, change, delete),
 3. the message length: is it correct for the type of file id and transaction type?
- read the file_id, transaction type and message length from the message
 - check that the file_id is within the limits of acceptable file IDs
 - is the transaction type one of the four types (add, update, change, delete)?
 - is the length correct given the file_id and transaction type

If any of these tests fail, `validate_transaction()` returns false, else it returns true.

26.5.8 FUNCTION: MNT_QUEUE_RE_WRITE

This just writes a message (back) to a local maintenance queue.

- lock the maintenance queue
- read the queue header. If reading the header fails then unlock the queue and exit
- check the validity of the header. Again, if this fails, unlock the queue and exit
- get the next available record index and increment it by the size of the record that we are about to write
- write the header back to the maintenance queue

- for each host attached to the queue increase its semaphore's count
- write the message to the queue
- unlock the queue

26.5.9 FUNCTION: PROCESS_TRANSACTION

process_transaction() is the central function of skymds24. Here each message, its 'file_id' having been determined, is passed to its appropriate handler. These handlers generally update <System_Name> database tables, though other activities are possible.

process_transaction() consists of only a huge (85 or so case) switch statement, the switch values and handlers called being listed in the following table.

<u>file_id</u>	<u>value</u>	<u>Constant</u>	<u>Description of Update</u>	<u>Function Called</u>
24	ID_HICCYMMDD		Primary history table update	<code>process_hist_rec()</code>
30	ID_HSCCYMMDD		Secondary history file update	<code>process_hsec_rec()</code>
28	ID_MDSPASTS		Pager stats master	<code>process_pasts_rec()</code>
29	ID_MDSCLSTS		Client stats master	<code>process_clsts_rec()</code>
87	ID_CHKPTN		Generic Checkpoint File	<code>process_rec()</code>
65	ID_FRCFM		FRC Facilities Manager File	<code>process_rec()</code>
66	ID_FRCET		FRC Enquiry Type File	<code>process_rec()</code>
67	ID_FRCEN		FRC Enquiry Code File	<code>process_rec()</code>
68	ID_FRCSI		FRC Site File	<code>process_rec()</code>
69	ID_FRCES		FRC Enquiry/Site Relationship File	<code>process_rec()</code>
70	ID_FRCTR		FRC Trade Code File	<code>process_rec()</code>
71	ID_FRCMA		FRC Maintenance Code File	<code>process_rec()</code>
72	ID_FRCSP		FRC Service Provider File	<code>process_rec()</code>
73	ID_FRCTS		FRC Trade/Site File	<code>process_rec()</code>
74	ID_FRCEL		FRC Enquiry Log File	<code>process_rec()</code>
75	ID_FRCML		FRC Maintenance Log File	<code>process_rec()</code>
76	ID_FRCQU		FRC Questionnaire File	<code>process_rec()</code>
19	ID_MDSADVIS		Advisory	<code>process_rec()</code>
3	ID_MDSAUDIT		Audit trail	<code>process_rec()</code>
15	ID_MDSBULL		Operator bulletin	<code>process_rec()</code>
64	ID_MDSCALLP		Call patching master	<code>process_rec()</code>
0	ID_MDSCLien		Client master	<code>process_line_client_rec()</code>
88	ID_MDSCMDET		Company details	<code>process_rec()</code>
17	ID_MDSCONTA		Contacts master	<code>process_rec()</code>
50	ID_MDSCTEXT		Escalation large canned text	<code>process_rec()</code>
62	ID_MDSCUSTP		Customer program	<code>process_rec()</code>
12	ID_MDSDELAY		Delay monitoring	<code>process_rec()</code>
54	ID_MDSDELST		Delivery Status File	<code>process_rec()</code>
1	ID_MDSDESTS		Destinations (queues)	<code>process_dests_rec()</code>
51	ID_MDSEMAIL		e-mail master	<code>process_rec()</code>
59	ID_MDSEMBLK		Email Queue File	<code>process_rec()</code>
60	ID_MDSAPARTY		A-Party Customer Info	<code>process_rec()</code>
49	ID_MDSDESTM		Destinations master	<code>process_destm_rec()</code>
22	ID_MDSESCAL		Escalation master	<code>process_rec()</code>
26	ID_MDSESTRN		Escalation transaction master	<code>process_rec()</code>
13	ID_MDSEXTCO		External contact master	<code>process_rec()</code>
14	ID_MDSALARM		Mdsalarm	<code>process_rec()</code>
63	ID_MDSFACIL		Facilities – <System_Name> special services	<code>process_facil_rec()</code>
27	ID_MDSFAX		Fax master	<code>process_rec()</code>
55	ID_MDSFMTXT		Holds curses	<code>process_rec()</code>
20	ID_MDSFORMS		Forms	<code>process_rec()</code>
36	ID_MDSFUNCT		Function codes	<code>process_rec()</code>
18	ID_MDSINPAK		Info pack.	<code>process_rec()</code>

Table 141: skymds24: Updates and Update Functions

<u>file_id</u>		Description of Update	Function Called
value	Constant		
4	ID_MDSLNE	Indial number information	<code>process_line_client_rec()</code>
6	ID_MDSNETCN	Network connection	<code>process_rec()</code>
21	ID_MDSNROST	Roster master	<code>process_rec()</code>
7	ID_MDSPAGER	Pager master	<code>process_pager_rec()</code>
10	ID_MDSPASSW	Password username	<code>process_rec()</code>
23	ID_MDSPHONE	Phone master	<code>process_rec()</code>
8	ID_MDSPORTS	Ports security	<code>process_rec()</code>
9	ID_MDSERROR	generic error	<code>process_rec()</code>
77	ID_MDSPUBH	Public holiday	<code>process_pub_rec()</code>
37	ID_MDSSECUR	Security codes	<code>process_secur_rec()</code>
33	ID_MDSSCODE	Service code	<code>process_rec()</code>
25	ID_MDSSEQU	Message sequence number	<code>process_rec()</code>
46	ID_MDSMSTA	Sms stats	<code>process_rec()</code>
82	ID_MDSSCHP	Search parameter	<code>process_rec()</code>
83	ID_MDSSCHD	Search data	<code>process_schd_rec()</code>
47	ID_MDSSRF	Software request form	<code>process_rec()</code>
53	ID_MDSREM	Reminders queue	<code>process_rem_rec()</code>
52	ID_MDSREMP	Reminders parameter	<code>process_remp_rec()</code>
56	ID_MDSTEXT	Text header	<code>process_rec()</code>
2	ID_MDSTIMEV	Timed Events table	<code>process_timev_rec()</code>
58	ID_MDSTMESG	Temporary message master	<code>process_rec()</code>
48	ID_MDSUNIQN	Unique number	<code>process_rec()</code>
61	ID_RENAME	Rename Text File	<code>process_rename_file()</code>
44	ID_STAA1	STARTEL Location	<code>process_rec()</code>
45	ID_STAA2	STARTEL Dealer	<code>process_rec()</code>
57	ID_TEXT	Flat text message	<code>process_text_file()</code>
78	ID_SELSE	Seeley Serial Number	<code>process_rec()</code>
79	ID_SELCO	Seeley Controller Type	<code>process_rec()</code>
80	ID_SELBR	Seeley Brand Code	<code>process_rec()</code>
81	ID_SELPR	Seeley Problem Code	<code>process_rec()</code>
16	ID_SELAD	Seeley Advisory Information	<code>process_rec()</code>
84	ID_SAFPR	Safety <Company_Name> Pro	<code>process_rec()</code>
85	ID_SAFSP	Safety <Company_Name> Service	<code>process_rec()</code>
86	ID_SAFLG	Safety <Company_Name> log	<code>process_rec()</code>
38	ID_VIPA1	VIP Data	<code>process_rec()</code>
39	ID_FRCGL	General Ledger Data	<code>process_rec()</code>
40	ID_FRCSC	Site/Cost Centre Data	<code>process_rec()</code>
41	ID_FRCCG	Call Centre/GL Data	<code>process_rec()</code>
43	ID_VIPA2	VIP Suburb	<code>process_rec()</code>
5	ID_MDSGL01	==== No Description available ===	<code>process_g1_rec()</code>

Table 142: skymds24: Updates and Update Functions (Continued)

26.5.10 FUNCTIONS: SKYMD24 HANDLER FUNCTIONS

Remember that `update_xxx()` functions re-write the entire record, while `change_xxx()` function only change particular parts of the record depending on a value in the maintenance queue message record (the ‘sub_type’ field). Also, in many of these handlers multiple implementation functions may be called, if one tries to change a non-existent record the trans_type may be altered to ADD_REC by the CHG_REC processing function, and the ADD_REC processor will then be used. All kinds of fun and games, roundabouts and swings.

In alphabetical order:

26.5.10.1 Function: process_clsts_rec

Process client stats. These will be added to (new record), changed (change a field, probably increment a call count) and occasionally deleted, but we should never need to update (re-write) an entire client record.

- use `set_rec_struct()` to copy the appropriate part of the message to a client stats record.
- depending on the transaction type (add, update, delete, or change), call the appropriate client stats manipulation Function:

tran_type	Client Statistics Handlers	Action
ADD_REC	<code>add_clsts()</code>	add a Client Stats record
UPD_REC	error: should never be re-writing a stats record	update a Client Stats record
DEL_REC	<code>delete_generic()</code>	delete a Client Stats record
CHG_REC	<code>change_clsts()</code> : increment call counters	change a Client Stats record

Table 143: skymds24: Client Statistics Handler actions

26.5.10.2 Function: process_destm_rec

Update the destinations master file.

- open the mdsdestm database table
- call `set_rec_struct()` to copy the message details into a destination master record

tran_type	Destinations Master Handlers	Action
ADD_REC	<code>add_generic()</code>	add a Destinations record
UPD_REC	<code>update_generic()</code>	update a Destinations record
DEL_REC	error. No deletions or changes	delete a Destinations record
CHG_REC	error. No deletions or changes	change a Destinations record

Table 144: skymds24: Destinations Master Handler actions

- close the destinations master table.

26.5.10.3 Function: process_dests_rec

A null handler: no action is taken. As per the comment in this (empty) function “The Destination files are not copied to other machines as they are unique to each individual machine”.

26.5.10.4 Function: process_facil_rec

This short function manages to add, update or change a range of <System_Name> facility record types. It does this in two (well three) steps:

- get the correct file pointer (table handle) to the facilities table (mdsfacil)

The facilities table is opened three times with three different indexes. The table handle with the correct index is chosen first

- use `set_rec_struct()` to copy the appropriate part of the message (its ‘detail’ structure) to the facilities record.
- depending on the transaction type (add, update, delete, or change), call the appropriate facilities manipulation Function:

tran_type	Facilities Handlers	Action
ADD_REC	<code>add_facil()</code>	add a Facilities record
UPD_REC	<code>update_facil()</code>	update a Facilities record
DEL_REC	<code>delete_facil()</code>	delete a Facilities record
CHG_REC	error. Must be deleted and re-inserted.	change a Facilities record

Table 145: skymds24: Facilities Handler actions

Each of these uses a `find_facil_rec()` function to locate the correct facilities record in the mdafacil facilities database table

26.5.10.5 Function: process_g1_rec

Add or changes a specialised record for the South Australian Government (SAGRN – South Australian Government Radio Network).

tran_type	SAGRN Handlers	Action
ADD_REC	<code>add_generic()</code>	add a SAGRN record
UPD_REC	error. Can't re-write the record	update a SAGRN record
DEL_REC	error. No Deletions.	delete a SAGRN record
CHG REC	<code>change_generic()</code>	change a SAGRN record

Table 146: skymds24: SAGRN Handler actions

26.5.10.6 Function: process_hist_rec

Here skymds24 writes primary history records, as supplied by maintenance queue messages. The primary history stores all the technical information (quite a lot) concerning each <System_Name> message.

- if the '-s' option has been used (don't do history updates), `process_hist_rec()` is exited
- The primary history table is then updated as per:

tran_type	Primary History Handlers	Action
ADD_REC	<code>add_hist()</code>	add a Primary History record
CHG_REC	<code>change_hist()</code>	update a Primary History record
UPD_REC	error. Can't update history: use change	delete a Primary History record
DEL_REC	error. Can't delete history	change a Primary History record

Table 147: skymds24: Primary History Updates Handler actions

The `change_hist()` function copies the existing secondary file indexes plus a few other bits and pieces to the new record and then writes this record over the existing record.

26.5.10.7 Function: process_hsec_rec

Similar to `process_hist_rec()`, `process_hsec_rec()` writes to the secondary history, a text file, not a database table. It does however insist that a matching primary history. The secondary history is the core information concerning the message – the message text, type of message and destination target – matching the parallel primary history record containing technical information.

- first `HIST_read_prim_equal()` is used to ensure that a matching primary history record exists
- The secondary history table is then updated as per:

tran_type	Secondary History Handlers	Action
ADD_REC	<code>add_hsec()</code>	add a Secondary History record
CHG_REC	<code>change_hsec()</code>	update a Secondary History record
UPD_REC	error. Can't update history: use change	delete a Secondary History record
DEL_REC	error. Can't delete history	change a Secondary History record

Table 148: skymds24: Secondary History Handler actions

`add_hsec()` wraps a call to `HIST_write_sec()` and `change_hsec()` similarly uses `HIST_update_sec()`

26.5.10.8 Function: process_line_client_rec

`process_line_client_rec()` add or changes the indial number associated with a particular client.

- open a database table.
- use `set_rec_struct()` to copy the appropriate part of the message (its ‘detail’ structure) to an indial record.
- if we are only processing one type of subscriber type (Sth Australian Govt only, SMS only, or <Company_Name> Only – no Optus, Vodafone...) then `subscriber_check()` is called to test that the record (in the client database table) is for a client with the correct subscriber type. This option is set up by the ‘-c’ commandline option.
- Now the appropriate handler function is called:

tran_type	Indial Update Handlers	Action
ADD_REC	<code>add_generic()</code>	add a Indial record
UPD_REC	<code>update_generic()</code>	update a Indial record
DEL_REC	<code>delete_generic()</code>	delete a Indial record
CHG_REC	<code>change_generic()</code>	change a Indial record

Table 149: skymds24: Indial update handler actions

to implement the indial number change. The contents of the message are the new record that the *_generic functions simply copy into the appropriate table.

26.5.10.9 Function: process_pager_rec

Add, update or change a pager record in the database.

- use `set_rec_struct()` to copy the appropriate part of the message (its ‘detail’ structure) to the pager record.
- depending on the transaction type (add, update, delete, or change), call the appropriate pager manipulation Function:

tran_type	Pager Record Handlers	Action
ADD_REC	<code>add_generic()</code>	add a Pager record
UPD_REC	<code>update_pager()</code>	update a Pager record
DEL_REC	error: deletions done by a clean-up program	delete a Pager record
CHG_REC	<code>change_pager()</code>	change a Pager record

Table 150: skymds24: pager record update handler actions

The updates and changes retain the call counts from the existing record.

26.5.10.10 Function: process_pasts_rec

These are pager stats, in case you wondered.

- use `set_rec_struct()` to copy the appropriate part of the message to a pager stats record.
- depending on the transaction type (add, update, delete, or change), call the appropriate pager manipulation Function:

tran_type	Pager Stats Handlers	Action
ADD_REC	<code>add_pasts()</code>	add a pager stats record
UPD_REC	error: never re-write an entire stats record	error
DEL_REC	<code>delete_generic()</code>	delete a pager stats record
CHG_REC	<code>change_pasts()</code> : increment call counters	change a pager stats record

Table 151: skymds24: pager statistics update handler actions

26.5.10.11 Function: process_pubh_rec

`process_pubh_rec()` manipulates records in the public holidays table (mdspubh).

- open the public holidays database table
- use `set_rec_struct()` to copy the appropriate part of the message to a public holidays record.
- depending on the transaction type (add, update, delete, or change), call the appropriate security code manipulation Function:

tran_type	Public Holidays Handlers	Action
ADD_REC	<code>add_pubh()</code>	add a public holidays record
UPD_REC	<code>update_pubh()</code>	update a public holidays record
DEL_REC	<code>delete_pubh()</code>	delete a public holidays record
CHG REC	<code>change_pubh()</code>	change a public holidays record

Table 152: skymds24: public holiday update handler actions

26.5.10.12 Function: process_rec

`process_rec()` is called as a response to 75% of the ‘file_ids’, the what-shall-I-update codes. It is the generic update function, calling the four `add_generic()`, `update_generic()`, `change_generic()`, and `delete_generic()` functions.

- opens the appropriate database table. The pointers / table handles were stored in an array during the skymds24 `initialise()` function
- use `set_rec_struct()` to copy the appropriate part of the message to a ‘rec’ structure
- depending on the transaction type (add, update, delete, or change), call the appropriate Function:

tran_type	Facilities Handler Function	Action
ADD_REC	<code>add_generic()</code>	add a record
UPD_REC	<code>update_generic()</code>	update a record
DEL_REC	<code>delete_generic()</code>	delete a record
CHG REC	<code>change_generic()</code>	change a record

Table 153: skymds24: generic processor handler actions

26.5.10.13 Function: process_rem_rec

`process_rem_rec()` adds, changes or deletes a reminder from the local <System_Name> site. Reminders are messages sent to Call Centre Operator’s telling them to perform certain tasks (at certain times), e.g. call a customer or send a fax

- open the reminder database table (mdsrem)
- use `set_rec_struct()` to copy the appropriate part of the message to a reminder record.
- depending on the transaction type (add, update, delete, or change), call the appropriate reminder manipulation Function:

tran_type	Facilities Handler Function	Action
ADD_REC	<code>add_rem()</code>	add a reminder record
UPD_REC	<code>update_rem()</code>	update a reminder record
DEL_REC	<code>delete_rem()</code>	delete a reminder record
CHG REC	<code>change_rem()</code>	change a reminder record

Table 154: skymds24: reminder update handler actions

26.5.10.14 Function: process_remp_rec

`process_remp_rec()` manipulates the reminders parameter file. As detailed above, reminders are messages sent to Call Centre Operator's telling them to perform certain tasks (at certain times), e.g. call a customer or send a fax.

- open the reminder parameter database table
- use `set_rec_struct()` to copy the appropriate part of the message to a reminder parameter record.
- depending on the transaction type (add, update, delete, or change), call the appropriate reminder parameter manipulation Function:

tran_type	Facilities Handler Function	Action
ADD_REC	<code>add_remp()</code>	add a reminder parameter record
UPD_REC	<code>update_remp()</code>	update a reminder parameter record
DEL_REC	<code>delete_remp()</code>	delete a reminder parameter record
CHG REC	<code>change_remp()</code>	change a reminder parameter record

Table 155: skymds24: reminder parameter update handler actions

26.5.10.15 Function: process_rename_file

As the name suggests this function renames a specific file in the local <System_Name> system.

- The existing file name and the new file name are read from the message record
- `txt_rename()` is used rename the file

26.5.10.16 Function: process_schd_rec

`process_schd_rec()` updates the 'search' database table, mdsschd. When many pagers are connected to the same indial, a service can be established such that the caller describes attributes of the person (pager holder) that they wish to talk to (First name, Area, and job title, say), the Call Centre Operator enters this data in a search form and the correct pager number is (hopefully) returned. The mdsschd database table holds data to assist the <System_Name> system in performing this search, and `process_schd_rec()` send updates to this table.

- use `set_rec_struct()` to copy the appropriate part of the message to a research record.
- depending on the transaction type (add, update, delete, or change), call the appropriate search table manipulation Function:

tran_type	Facilities Handler Function	Action
ADD_REC	<code>add_schd()</code>	add a search record
UPD_REC	<code>update_schd()</code>	update a search record
DEL_REC	<code>delete_schd()</code>	delete a search record
CHG REC	<code>change_schd()</code>	change a search record

Table 156: skymds24: search record update handler actions

26.5.10.17 Function: process_secur_rec

Update the Security Codes database table (mdssecur).

- open the security code database table
- use `set_rec_struct()` to copy the appropriate part of the message to a security code record.
- depending on the transaction type (add, update, delete, or change), call the appropriate security code manipulation Function:

tran_type	Facilities Handler Function	Action
ADD_REC	<code>add_secur()</code>	add a security code record
UPD_REC	<code>update_secur()</code>	update a security code record
DEL_REC	<code>delete_secur()</code>	delete a security code record
CHG REC	<code>change_pasts()</code>	change a security code record

Table 157: skymds24: security code update handler actions

- each of these manipulation functions calls `update_secur_shared_memory()` to update the local security shared memory block, making the changes instantly applied.

26.5.10.18 Function: process_text_file

`process_text_file()` writes the contents of the message to a text file, making an easy way to transmit text-files between <System_Name> sites.

- reads the first TEXT_DIR_SIZE-1 characters as the filename
- opens the file with the `TXT_open_file()` function
- writes the message ‘detail’ to the file, which may be any text.
- close the text file with `TXT_close_file()`

26.5.10.19 Function: process_timev_rec

This short function manages to add, update or change a range of <System_Name> timed-events record types. It does this in two (well three) steps:

- get the correct file pointer (table handle) to the timed-events table (mdstimev)

The timed-events table is opened three times with two indexes, so the first task is to choose the correct table handle.

- use `set_rec_struct()` to copy the appropriate part of the message (its ‘detail’ structure) to the timed-events record.
- depending on the transaction type (add, update, delete, or change), call the appropriate timed-events manipulation Function:

tran_type	Facilities Handler Function	Action
ADD_REC	<code>add_timev()</code>	add a Timed Event record
UPD_REC	<code>update_timev()</code>	update a Timed Event record
DEL_REC	<code>delete_timev()</code>	delete a Timed Event record
CHG_REC	error. Must be deleted and re-inserted.	error

Table 158: skymds24:timed events update handler actions

Each of these uses a `find_timev_rec()` function to locate and validate the correct timed-events record in the mdstimev timed-events database table. Because of the many types of timed-events this is actually more involved than it sounds, with reasonably extensive validity testing. A separate `find_safety<Company_Name>_timeve_rec()` function is used to locate the Safety<Company_Name>³⁹ timed events.

26.5.10.20 Functions: add_generic, update_generic, change_generic, and delete_generic

These functions are used by a number of update functions to provide standardised updates.

Function	Description
<code>add_generic()</code>	add an arbitrary record to an arbitrary table. If a matching record already exists adjust the trans_type to CHG_REC and return.
<code>update_generic()</code>	try twice to read the existing record. If this succeeds, write the new record over the top of the old, else change the trans_type to ADD_REC and return.
<code>change_generic()</code>	change an arbitrary record in an arbitrary table. If this succeeds, write the new record over the top of the old, else change the trans_type to ADD_REC and return.
<code>delete_generic()</code>	read the desired record from the desired database table and delete it

Table 159: skymds24: generic function descriptions

26.5.11 FUNCTION: DISPLAY_MNTQ_REC

`display_mntq_rec()`, called while in debug mode, writes a formatted version of the current maintenance queue message to stderr.

³⁹ Safety<Company_Name> is a system of tracking workers in dangerous environments, where they have to make periodic “I’m OK” calls to a <Company_Name>Net call centre to indicate that they are, in fact, OK.

26.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

26.7 STOPPING SKYMDS24

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)

27 skymds28

27.1 SKYMD28 OVERVIEW

skymds28 establishes the connection of a non-local <System_Name> application to either the recent history data on another <System_Name> system or the history server (machine) in Melbourne. You can think of skymds28 as the history server (daemon) for whatever machine it is on.

An application, needing some historical data, calls a history access function which decides whether the data is to be found locally, in another site's recent-history collection, or on the Melbourne history server. If the data is local it is accessed directly. If the data is not local the application connects to a newly-spawned instance of skymds28 running on another <System_Name> server (recent data) or on the history server in Melbourne. skymds28 obtains the data from the history server and passes it back to the caller.

27.2 RUNNING SKYMD28

When running the <System_Name> start-up script start.mds the user is prompted thus:

```
Start skymds28, skymds81, skymds06, skymds27, remterm and  
skymds20 (y/n) :
```

Presuming that he or she answers 'y', a skymds28 daemon is executed. When this daemon receives a request for history data it forks itself, the child handling specific data retrieval, returning the data, and terminating.

27.2.1 COMMAND LINE PARAMETERS

Parameter	Description
Mandatory	
-t < device / port >	The port against which skymds28 is to be registered in the mdsports database table
-p < port >	the port over which communication is made and data sent from and to the calling process
Optional	
-D	run in debug mode, displaying detailed run-time information
-?	print usage instructions and exit.

Table 160: skymds28: Command-line Parameters

27.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute skymds28 at one <System_Name> site is:

```
$MDS_BIN_DIR/skymds28 -p 2033 -t /dev/dum20 2>$LOG_SCR
```

indicating that skymds28 is run:

- communicating and sending data over TCP/IP port 2033
- registered against the device /dev/dum20 in the mdsports database table
- duplicating its stderr stream to the object LOG_SCR, defined in start.env to be /dev/ttyle0, a terminal somewhere.

27.3 PROGRAM DESCRIPTION

skymds28 is a server for history data, both on <System_Name> sites and on the lnkh History Server in Melbourne spawning copies of itself to manage the history access. Most of the actual work is done by the various HIST_* and NHIST_* functions coded in mdshisio.c: both skymds28 and the program requesting the history data use these functions to accomplish their tasks. skymds28 is more like a 'focus' or a 'conduit' for the history requests, receiving these requests via HIST_*() function calls, and then using these functions to actually retrieve the data.

The `main()` function passes control to `process_contact()` each time a history request arrives. `process_connect()`, a switch-board function, passes control the appropriate request-handling function (e.g. `process_prim_read()`) each of which are wrappers for `HIST_*()` functions (e.g. `HIST_read_prim()`). These `HIST_*()` functions write to or read from the appropriate history data.

27.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds28.

27.4.1 SOURCES FILES

Source File	Description
alarmrtn.c	Generic alarm routine.
gensubs.c	Contains various general-purpose routines that are separately compiled.
globvar.c	Functions to access the Global Variable shared memory used by multiple programs.
isstart.c	Wrapper for C-ISAM isstart function – choose index and position the read-point
mdscdr.c	Contains various general purpose routines that are separately compiled.
mdsemqmain.c	Generic message email queuing routine.
mdsginit.c	A number of functions, which perform commonly, used initialisation code segments.
mdshisio.c	Generic history file handling routines.
mdskbld.c	Contains functions to build and unbuild pager keys, and set-up a dummy pager.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurt.c	Generic message queuing routine.
mdssemcck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdstmrtn.c	Contains functions, which perform operations to do with time.
skymds28.c	Network CISAM file server.

Table 161: skymds28: Source Files

27.4.2 HEADER FILES

Header File	Description
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.: C time field definitions
globvar.h	Global shared memory structures
macros.h	User defined macros.: definitions & macros for common functions
mdsalrec.h	MDSALARM definition file
mdsaprec.h	A-Party Master file
mdscdr.h	CDR definitions file
mdscdrec.h	CDR data structure
mdsderec.h	Ports security master file
mdsfldsz.h	Standard <System_Name> includes.: field length & definitions
mdsg1rec.h	GL3000 tnpp record: <System_Name> □ GL3000 CDR record layout
mdsg2rec.h	GL3000 Billing record: GL3000 □ <System_Name> CDR record layout
mdshirec.h	History primary recs. file: New History logging file
mdshrec.h	History server transaction layout
mdshsrec.h	Secondary History file (audits)
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsncrec.h	Network Connection master file
mdsparam.h	parameters used by PMS system
mdsparec.h	Pager Master File
mdsporec.h	Ports security master file: Ports Record
mdspsec.h	Password master file
mdsquarec.h	Contacts Message queue file: Destination Message queue file
mdstrec.h	MNTQ: maintenance queue file

Table 162: skymds28: Header Files

27.4.3 DATABASE TABLES

Each of these header files contains (only) a C-struct that matched the structure of a database table used by skymds28.

Database Table	Header File	Description
mdsaparty	mdsaprec.h	A-Party Customer Info
mdscdrec	mdscdrec.h	Call details master
mdsdestm	mdsderec.h	Destinations master
mdsgl01	mdsglrec.h	Record layout of message from <System_Name> to a TNPP site
mdsgl02	mdsg2rec.h	gl3000 ver 6.00a billing protocol
hiccyymmdd	mdshirec.h	the 'logical' history
hsyymmdd	mdshsrec.h	History secondary storage
mdsline	mdslirec.h	Indial number information
mdsnetcn	mdsncrec.h	Network connection
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdspassw	mdpsrc.h	Password username
mdsqxx.txt	mdsquarec.h	Destination queues
mdsqa1.txt	mdstrrec.h	Inter processor transaction queue

Table 163: skymds28: Database Tables

27.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description	Example
MDS_MACHINE_ID	What machine we are running on	'M' (Melbourne)
MDS_DATA_DIR	Where the database tables are	/opt/mds/data/
MDS_BIN_DIR	Where the <System_Name> executables are	/opt/mds/bin/

Table 164: skymds28: Environment Variables

27.5 SKYMD28: MAJOR FUNCTIONS

27.5.1 FUNCTION: MAIN

skymds28's `main()` waits for data (history updates or reads) and "forks" when one is received. After forking:

- The Child: calls `process_connect()` to action the history update or read, after which the child terminates.
- The Parent: closes the socket, waits for the child to finish processing (using the `wait()` system function) and loops to receive the next history request (and spawn another child)

Just in-case you aren't aware, "fork"-ing occurs when an application makes complete, running, copy of itself. The two processes test to see which they are ('parent' or 'child') and act appropriately, presumably different. In our case the parent does nothing, the child processes the history data, and terminates.

The slightly longer description of skymds28's `main` is:

- calls `initialise()` to prepare the run-time environment

Now the infinite loop (get a history request, process it) starts.

- test the shut-down semaphore: time to die? This is the intended exit point of the loop, `main()`, and skymds28.
- wait for the next history request. The `accept()` system call 'blocks' (doesn't return) until some communication is detected over the socket it is listening on.

skymds28 now forks. As noted above:

- The Child: calls `process_connect()` to action the history update or read, after which the child terminates.
- The Parent: closes the socket, waits for the child to finish processing (using the `wait()` system function) and loops to receive the next history request (and spawn another child)

27.5.2 FUNCTION: INITIALISE

The initialise() function sets up skymds28's run-time environment, via the following steps:

- Reads in the required environment variables
- parses the command-line
- calls `background()` to run skymds28 as a background daemon.
- catch a few signals
- use `PORT_set_port_started()` to record in mdsparts that skymds28 has started
- obtain the number of the shutdown semaphore
- finally call `set_up_listen_socket()` to prepare the socket to listen for data requests on.

27.5.3 FUNCTION: PROCESS_CONNECT

`process_connect()` controls the retrieval and / or updating of history data. As soon as skymds28 spawns a child copy of itself (after seeing either a history data request or update waiting for processing) the child passes control here, to `process_connect()`. `process_connect()` uses `get_next_mess()` to extract the message from the socket on which it is listening to the data port, determines what type of history action is required, and finally calls the appropriate handler.

- first `process_contact()` forks itself (yes, again) and kills the parent. This puts it in the background. It also closes its inherited copy of the socket that its parent used to listen for messages on.
- Now the maintenance queue is prepared.

The remainder of the function is enclosed in (another) infinite loop which reads each history request or record from the socket (with `get_next_mess()`), processes them, finally exiting when the request are all serviced and the records all filed

- `get_next_mess()` reads the data from the socket. It tries this three times, finally failing if there is nothing there.
- now the appropriate handler is called.

Internal Constant	Description	Handler:	Which calls:
ISREAD	Read a primary history record	<code>process_prim_read()</code>	<code>HIST_read_prim()</code>
ISWRITE	Write a primary history record	<code>process_prim_write()</code>	<code>HIST_write_prim()</code>
HS_READ_SEC	Read next secondary history	<code>process_hs_read_next()</code>	<code>HIST_read_sec_next()</code>
HS_WRITE_SEC	Write next secondary history	<code>process_hs_write()</code>	<code>HIST_write_sec()</code>
ISWRITE_RETRV	Store history read by operator	<code>process_write_retrieval()</code>	<code>HIST_write_retv_to_hsec()</code>
SET_MODE_TRANS		<code>process_set_mode()</code>	

Table 165: skymds28: Top-level History Request Handlers

27.6 TOP LEVEL HISTORY REQUEST HANDLERS

These follow a uniform pattern, calling a `HIST_*()` function to read from or write to the history file, formatting a response, and sending that response back to the calling function. All these functions are coded in skymds28.c.

27.6.1 FUNCTION: PROCESS_PRIM_READ

Here we read a primary history record from whatever history file we are connected to. The primary history is all the information *about* the messages while the secondary history contains the message itself.

- call `HIST_read_prim()` to actually read the data from the history. `HIST_read_prim()` is another switch-board function, passing control to one of nine specific history retrieval functions to do the actual retrieval.
- Add a few bits of data to the retrieved history record.
- write the record to socket

27.6.2 FUNCTION: PROCESS_PRIM_WRITE

Write a primary history record, info about a call.

- HIST_write_prim() is used to actually write the primary history record. See the description below.
- next the reply structure is filled. This is just a couple of fields (duplicate found, read/write) though, if the history record written was detected to be a duplicate, the entire record, up to 2 kb, is added.
- The reply structure is sent back to the calling process, written to the socket. The calling process could be on another <System_Name> site.
- Finally, the history write is copied to the other servers in the same city, i.e. the second servers in Melbourne and Sydney. This is achieved by writing the data to the maintenance queue.

27.6.3 FUNCTION: PROCESS_HS_READ_NEXT

process_hs_read_next() reads a secondary history record which contains the message text itself. The secondary history is held as a text file (one for each day), while the primary is stored as a database table, (one for each day).

What happens (what must happen) in practice is this:

1. A primary history record is obtained. This is the core information about the call.
2. HIST_posn_sec() is called to get to the first secondary record relevant to the primary record
3. process_hs_read_next() (or HIST_read_sec_next() directly) is called to retrieve each of the other secondary history records matching the primary history record.

There may be multiple secondary records for each primary record. One or more secondary history records are constructed when the call is first placed and, following that, each time the message is retrieved, used in a report, resent, or in any other way interacted with a further secondary history record is made.

So, given that the primary history record has been obtained and HIST_posn_sec() (and check_hist_and_sec()) called, process_hs_read_next() proceeds in the following manner.

- HIST_read_sec_next() is called to read the appropriate record from the secondary history file into a HRREC transaction record (defined in mdshrrec.h).
- A few additional details are written to the record
- The transaction record is written back to the calling process, with or without a secondary history record included.

27.6.4 FUNCTION: PROCESS_HS_WRITE

This function writes a secondary history record to the secondary history file.

- both the primary history record (existing) and the second history record (to be written) are passed to HIST_write_sec(), which stores the new secondary history record.
- a HRREC record is populated and written back to the socket connecting skymds28 to the application writing the history data.
- finally the records are written to the maintenance queue, for update to either other <System_Name> machines in the same city, or to all <System_Name> sites, depending on the type of secondary history record.

27.6.5 FUNCTION: PROCESS_WRITE_RETRIEVAL

process_write_retrieval() records (in the secondary history) that an operator or process has read a record from the history table.

- first reply back to the calling application that the write has been done. This speeds things up for the external process: it doesn't have to wait while we mess with the database
- now call HIST_write_retrv_to_hsec() to record that a retrieval has been done.

27.6.6 FUNCTION: PROCESS_SET_MODE

process_set_mode() changes (or affirms) the history retrieval ‘mode’, toggling between ‘ONLINE’ mode; accessing from the local history, and ‘ARCHIVED’; accessing data up to a year old from the history server in Melbourne.

- The change in mode is implemented by calling the HIST_start_session(). This re-initialises skymds28, possibly into a new retrieval mode
- A message is written back to the external application indicating that the mode has been changed

27.7 LOWER LEVEL ‘UTILITY’ HISTORY FUNCTIONS.

These functions are called by skymds28’s process_*() functions described above *and* by the many <System_Name> applications that access the local history files directly.

Program	Description
cusapd01.c	Customised Service for APD.
cussta03.c	Send to X or <COMPANY_1_NAME> a fax of the details of the call processed by the customised <Company_1_Name> utility.
mdshismer.c	Overnight History Merging File
skymds03.c	Message delivery to all (current) delivery systems
skymds20.c	Extract messages from the bulk queue (mdsqa0.txt) and put them in the real queues.
skymds24l.c	Get messages from the network (from skymds23l) and use these transactions to update the files.
skymds24s.c	Get messages from the network (from skymds23l) and use these transactions to update Oracle DB.
skymds27.c	Processes contacts(Rosters Escalations TMS etc) from the contacts queue file.
skymds58.c	Recalculate undelivered messages for TMS.
skymds46a.c	Event Message Report.
skymds46b.c	Contact Message Report.
skymds46f.c	Send to the Fax/Email a message with the complete layout of a form

Table 166: skymds28: non-skymds28 programs using history functions

In addition to the HIST_*() files described below, a range of NHIST_*() functions are used by these other, non-skymds28 applications. Both the HIST_*() and NHIST_*() functions are coded in mdshisio.c.

27.7.1 FUNCTION: HIST_START_SESSION

HIST_start_session(), coded as a utility function in mdshisio.c, initialises and re-initialises access to the <System_Name> history system and is called by all programs needing to read from or write to the history system.

- read the global shared memory into a GLOBVAR structure

the global shared memory contains only data about the number of days of history are stored in various locations (locally, archived (compressed) and on the history server).

- call HIST_close_current_hist_and_hsec() to close the history files if they are open.

HIST_start_session() is called by systems currently interacting with the history server to change their ‘mode’, whether they are reading archived history. So this initialisation function needs to close any open history files.

- set the ‘mode’ to whatever has been passed in: ARCHIVED or ONLINE.

27.7.2 FUNCTION: CHECK_HIST_AND_SEC

check_hist_and_hsec() checks the required history files exist and can be opened.

- if the history file being tested is not today's, use HIST_close_current_hist_and_hsec() to close the history file.
- if the date of the history file being requested / tested is older than the oldest, change the request such that it requests the oldest file.
- use HIST_open_hist_and_sec() to ensure that the primary and secondary history files requested can be opened.

If the history file requested could be opened return TRUE, else return FALSE.

27.7.3 FUNCTION: CHECK_HIST_AND_SEC

As noted before, check_hist_and_hsec() check the required history files exist and can be opened.

- if the history file being tested is not today's, use HIST_close_current_hist_and_hsec() to close the history file.
- if the date of the history file being requested / tested is older than the oldest, change the request such that it requests the oldest file.
- use HIST_open_hist_and_sec() to ensure that the primary and secondary history files requested can be opened.

If the history file requested could be opened return TRUE, else return FALSE.

27.7.4 FUNCTION: GET_NEXT_MESS

get_next_mess(), coded in skymds28.c, reads, into a character buffer, the next history request from an external process. In a loop, up to three times, (three attempts to read a history request), get_next_mess():

- tries to read the port over which a socket connection to an external process has been established,
- if this is the third unsuccessful attempt then exit
- sleeps for one second if there is nothing to read on the socket

out of the loop: we have read something from the external application

- append a '\n' character to the end of the string.

27.7.5 FUNCTION: HIST_READ_PRIM [PRIMARY HISTORY]

HIST_read_prim() consists of on switch statement passing control to one of nine specialised functions reading the primary history. The usage here of the word 'mode' as in "shall I read the first, next, last ... record" is different from the other usage, as in read from the archive (archive mode) or locally (online mode).

Constant 'mode'	Function	Description
ISFIRST	<code>HIST_read_prim_first()</code>	reads the first record
ISLAST	<code>HIST_read_prim_last()</code>	reads the last record
ISNEXT	<code>HIST_read_prim_next()</code>	reads the next record
ISPREV	<code>HIST_read_prim_prev()</code>	reads the previous record
ISEQUAL	<code>HIST_read_prim_equal()</code>	reads the equal value
ISGREAT	<code>HIST_read_prim_great()</code>	reads the greater value
ISGTEQ	<code>HIST_read_prim_gteq()</code>	reads the >= value
ISLTEQ	<code>HIST_read_prim_lteq()</code>	reads the <= value
ISLESS	<code>HIST_read_prim_less()</code>	reads the < value

Table 167: skymds28: Specialised Primary History reading functions

27.7.6 9 FUNCTIONS: SPECIALISED PRIMARY HIST. READ FUNCTIONS [PRIMARY HISTORY]

These nine functions, performing the tasks as described in the table above, all have a similar general structure. Something like the following:

- call check_hist_and_hsec() which checks the required history files exist and can be opened. The dates are recorded in the HIREC structure passed into check_hist_and_hsec().
- use the Informix C-ISAM isread() function to read the a record from the appropriate history database table

If we have to search for a record, or read multiple records, a loop is started to read the collection of records. If it is a matter of finding one single possible record, then we have either found it or we haven't: so exit.

- Working on whatever history record we currently have, use validate_hist_rec() to ensure that the line number, id_type, and pager contact id are all present on the history record. If they are, then we are done: exit.
- Now move forward or backwards in time or through the records satisfying our criteria
- Have we moved to the previous or next day? If so check that we haven't gone too far (like into the future) and open another day's history table.
- again call check_hist_and_hsec() to check the required history files exist and can be opened.
- read the next record

Go around the loop again.

27.7.7 FUNCTION: HIST_WRITE_PRIM [PRIMARY HISTORY]

As the name suggests this function writes to the primary history file.

- call check_hist_and_hsec() to check the required history files exist and can be opened.

The remainder of the function is mostly concerned with what to do if a duplicate is encountered. As the index often used is based only on indial-number and time, a call initiating a contact-type advanced service may trigger multiple history records at the same time (for the same indial). These are handled by advancing the time by one second until the time is unique for this indial.

A loop (MAX_PRIM_DUPLICATES = 20 long) is started.

- attempt to write the history data to the primary history database table. If the write is OK, exit the loop.
- if a duplicate then add one second to the history record's timestamp. If we have passed midnight then move onto the next day's history file. Either way, start the loop again
- If there was some other error return with an error.

End of the loop

- now handle the situation where the maximum number of duplicates was exceeded: 20 history records for one indial on 20 consecutive seconds. Probably doesn't happen too often

27.7.8 FUNCTION: HIST_POSN_SEC [SECONDARY HISTORY]

A simple function that writes the length and the index (position) of the desired secondary history record into a structure used to look the record up in the secondary history table. These are both simply copied from the relevant primary history record. Two instructions, two lines of code, three lines of description.

27.7.9 FUNCTION: HIST_READ_SEC_NEXT [SECONDARY HISTORY]

Read the (next) history record from the secondary history text file.

- using the index and length determined in the previously described HIST_posn_sec() to generate a pointer to the correct record in the secondary history text file
- read the record
- check the magic number, which is currently 173, the ASCII character '{'. This is used to ensure that we have aligned the history read correctly

27.7.10 FUNCTION: HIST_WRITE_SEC [SECONDARY HISTORY]

Here skymds28 writes a record to the secondary history text file.

- If we need to use change the history files we are dealing with, use check_hist_and_hsec() to ensure that the required history files exist and can be opened.
- add a few bits of standard information to the record to be written
- lock the secondary history text file.
- write the record to the end of the history text file
- increase the ‘last secondary history’ index in the related record in the primary history and, in if this is the first secondary record attached to the primary record, set the ‘first secondary history’ index in said primary record.
- test that the record just written can be read.

done

27.7.11 FUNCTION: HIST_WRITE_RETRV_TO_HSEC [SECONDARY HISTORY]

Every time a retrieval from history is done a history record must be written. HIST_write_retrv_to_hsec() writes these “audit trail” records to the secondary history

- first prepare the record to write. There is quite a lot of preparation

Now a loop is started. One secondary history record is written for each retrieval. This loop iterates over the retrievals, writing a record for each.

- HIST_write_sec() writes the record to the secondary history
- the record just written is copied to the maintenance queue, for duplication to all centres if the record originally retrieved was a contact, else just to the other <System_Name> server in the same city (if there are two in the same city, as is the case in Sydney and Melbourne)
- Exit if the date we are dealing with is now in the future
- Else, if there is another relevant primary history record (e.g. a group paging sent five messages), move onto the next relevant primary record
- Go around the loop again.

27.8 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

27.9 STOPPING SKYMDS28

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)

<System_Name>: Distribution Programs

28 skymds03

28.1 SKYMDS03 OVERVIEW

skymds03 manages the delivery of <System_Name>'s messages to a range of external destinations. 30 to 40 (differently configured) instances of skymds03 run at any one time on a <System_Name> installation, sending messages of many different types (fax, email, pager messages,...) to an arbitrary number of destinations. skymds03 includes 10 sub-modules when built, and, after the huge call-centre message entry system, skymds03 is the largest program in the <System_Name> suite.

Currently skymds03 sends messages to the following destinations:

- <System_Name> Encoders. These messages are delivered as pager messages.
- The Fax server(s)
- Email servers
- SMSCs, (Short Message Service Centers), for the delivery of SMS text messages.
- SAGRN: The South Australian Government Radio Network
- External paging systems. Some messages are forwarded from <System_Name> to external parties' paging systems
- forwarding messages to instances of skymds03 running on <System_Name> systems in other cities

To actualise these deliveries, skymds03 uses multiple protocols. These include:

- TNPP: Telocator network paging protocol
- XACOM protocol – for interface to XACOM brand devices
- PET: Pager entry terminal protocol
- SMPP: Short message peer to peer protocol
- TCP/IP: Transmission control protocol/internet protocol

28.2 RUNNING SKYMDS03

The 30 – 40 instances of skymds03 are all triggered, like most <System_Name> daemons, from the start.mds start-up script. The user / administrator / supervisor is prompted:

Start skymds03 (y/n) :

upon which an answer of 'y' leads to a raft of skymds03s being launched.

28.2.1 COMMAND-LINE PARAMETERS

Parameter	Description
<u>Mandatory</u>	
-t < port >	the port against which this instance of skymds03 is registered in the mdsports database table
-d < queue >	The destination queue mnemonic. Where this skymds03 will deliver its messages.
<u>Optional</u>	
-P < length >	the maximum length of a pager number when sending a PET protocol message.
-D	run in debug mode, printing detailed run-time information to a log screen.
-n	Don't update the <System_Name> history system with the time each message was sent.
-s	"Info Pager": don't actually send pager messages to an encoder.
-b < count >	How many times we should pretend we received a reply from the encoder when we didn't.
-l < port >	The TCP/IP port on which skymds03 should listen.
-c	Attach the real pager number (instead of any substitute pager number) to PET transmission.
-C	Never time-out waiting for a PET connection.

Table 168: skymds03: Command Line Parameters

28.2.2 EXAMPLE COMMAND-LINE

Three examples of skymds03 command lines follow. Almost all (98%) as per the first example, with examples 2 & 3 being deviants. Most of the command-line options are not used.

```
$MDS_BIN_DIR/skymds03 -t /dev/mlb26 -d MEL5 -l 3014      2>$LOG_SCR  
$MDS_BIN_DIR/skymds03 -t /dev/dum044 -d BULKS -n          2>$LOG_SCR  
$MDS_BIN_DIR/skymds03 -t /dev/vcm47 -d NZL1 -l 3191 -P 7  2>$LOG_SCR
```

Example: P1 runs skymds03:

- registered against port /dev/mlb26 in the mdsports database table
- sending messages to the MEL5 queue, Melbourne's pager queue #5
- listening on the TCP/IP port 3014
- copying stderr messages to LOG_SCR, defined in start.env to be `/dev/tty0p5`

Example: P2 runs skymds03:

- registered against port /dev/dum044 in the mdsports database table
- sending messages to BULKS, the Bulk Queue (don't know why its doing that?)
- not updating the <System_Name> history messages with the time messages were sent.
- copying stderr messages to LOG_SCR, defined in start.env to be `/dev/tty0p5`

Example: P3 runs skymds03:

- registered against port /dev/vcm47 in the mdsports database table
- sending messages to the NZL1 queue, for delivery to New Zealand
- listening on the TCP/IP port 3191
- Setting the longest allowable pager number to be 7 digits.
- copying stderr messages to LOG_SCR, defined in start.env to be `/dev/tty0p5`

28.3 PROGRAM DESCRIPTION

skymds03 is the <System_Name> message delivery program, handling, with different modules, delivery of all <System_Name> message types to all destinations. The top-level logic is contained in skymds03's main() function which, in a loop,

1. gets the next message to send, which may come from a skymds03 program at some other <System_Name> site
2. get a remote message or get a local message. Remote messages have priority.
3. send the message with the function pointed to by the (mess_to_encoder*)() function pointer.
4. handles the reply, which most often includes updating the queue that the message came from.

The function pointer (mess_to_encoder*)() is pointed to one of 13 processing functions, depending on the protocol field of the record in the mdsdests database table that has a matching destination queue as that passed on the command-line. The correct processing function, coded in a sub-module, manages, processes and delivers the specific message in the correct manner and to the correct location..

28.4 PROGRAM STRUCTURE

28.4.1 SOURCE FILES

The program is made up of multiple modules, which are defined separately in source files. The following is a list of all these source files:

Source file	Description
skymds03.c	Message delivery to a single encoder
mdssendm.c	Message delivery to merp computers
mdssendx.c	Functions for message delivery to XACOM encoder with reply
mdssendw.c	Message delivery to XACOM encoder with single char
mdssendd.c	Message delivery to Dummy XACOM Encoder
mdssendp.c	Message delivery to dial up PET receivers
mdssendl.c	Message delivery via LAN to another skymds03
mdssendif.c	Faxing Initiator for both BATCH & ONLINE
mdssende.c	Email Initiator for both BATCH & ONLINE
mdssendv.c	Message delivery from Tone & Voice to reminders
mdstamp.c	Date time stamp & sequence number generation
crcalc.c	Calculate the CRC for a given string.
gensubs.c	Contains various general purpose routines that are separately compiled
isstart.c	Convert the key definition string into the c-ISAM structure
mdslog.c	Generic error logging
mdsemck.c	Check the nominated semaphore and return TRUE if it has dropped
mdsqmnt.c	Queues file updates for x mission to other machines
mdscomio.c	Various serial i/o routines
mdsrem.c	Functions to access the reminders queue
form_path.c	Forms a full pathname by concatenating the environment variable to the name passed
spawn.c	Spawns off a child process and (optionally) waits
mdscstats.c	Read the client statistics file for the line number
mdsginit.c	A number of functions which perform commonly used initialization code segments
globvar.c	Functions to access the Global Variable shared memory
mdsportn.c	Routines associated with the ports file
mdspstats.c	Read the pager statistics file for the pager number
mdsqusub.c	Various functions which are generic to the message
mdssetup.c	This module contains routines to obtain its setup definitions from a text setup file
mdscdr.c	Contains various general purpose routines
mdstxrtn.c	A number of functions which perform commonly used
mdsemqmain.c	Generic message email queuing routine
mdsqrtn.c	Generic message queuing routine
mdskbld.c	Contains functions to build and un build pager keys
mdssenda.c	Delivery message to MTP Server
mdsmtp.c	MTP Specific functions
mdscomm.c	Socket IO and Serial module
tnppcrc.c	Calculate the CRC for a given TNPP string
mdsphol.c	Functions to attach, check and modify Public Holiday
mdssendz.c	Message delivery to SMSC with reply.
mdssendn.c	Message delivery to TNZ via PACNET X.25
mdstmrtn.c	Contains functions which perform operations to do with time
mdshisio.c	Generic history file handling routines
mdsqmain.c	Generic message queuing routine
alarmrtn.c	Generic alarm routine
mdsunixos.c	Unix OS parts
mdsemenc.c	Contains functions which perform encoding message or files into an ANSI file

Table 169: skymds03: source files

28.4.2 HEADER FILES

The program uses multiple data types, macros, and database tables, which are all defined in header files. The following is a list of these header files:

Header file name	Description
gettime.h	C time field definitions.
macros.h	User defined macros.
gentypes.h	User defined types and Boolean
mdsflsz.h	Various field length and field definitions used by the MDS system
mdsparam.h	Various parameters used by the PMS system
mdssmpp.h	Definitions and packet descriptions for the SMPP protocol
mdsphol.h	Public Holiday Shared Memory file
mdscdr.h	CDR definitions.
mdsderec.h	Destinations master file
mdshirec.h	New Primary history file
mdshsrec.h	New Secondary history file
mdsparec.h	Pager master file
mdsporec.h	Ports security master file
mdsquarec.h	Message queue file
mdslirec.h	Line master file
mdsaprec.h	Aparty master file
mdsfxrec.h	Fax master file
mdsemrec.h	Email All master file
mdsrerec.h	Record reply back to remote host
mdsncrec.h	Network connection file
mdsacrec.h	Alternate CDR file

Table 170: skymds03: header files

28.4.3 DATABASE TABLES

The program accesses multiple database tables, which are all defined in header files. The following is a list of these database tables:

Table name	Header file	Description
mdsaparty	mdsaprec.h	A-Party Customer details
mdsdestm	mdsderec.h	Destinations master
mdsemail	mdsemrec.h	Email master
mdsfax	mdsfxrec.h	Fax master
hiccyymmdd	mdshirec.h	the 'logical' history
hsyymmdd	mdshsrec.h	History secondary storage
mdsline	mdslirec.h	Line – indial
mdsnetcn	mdsncrec.h	Network connection
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsquarec.h	Destination queues

Table 171: skymds03: database tables

28.5 SKYMD01: MAJOR FUNCTIONS

28.5.1 FUNCTION: MAIN

skymds03's `main()` function contains its central processing loop i.e. control is not passed off to some `do_it()` sub-function.

- call `initialise()` to set-up skymds03's run-time environment.
- initialise a few flags

Now the infinite loop starts. Here we obtain and process each message for delivery.

- Test the shut-down semaphore. Time to die?
- We may have got back to the start of the loop by there being no messages. If that is the case we do a few tests and call `wait_for_either_input()` to get the next message
- as long as the flag `send_remote_message` is still true, skymds03 now calls `get_next_remote_mess()` to obtain a message from a skymds03 program running on another <System_Name> site.

After getting the message we use `is_message_duplicated()` to check, well, you can guess, and then send an appropriate status indicator back to the other, distant skymds03 (via the `reply_to_host()` function). If we did receive an foreign message we skip the next 'local message' retrieval.

- Now, if there were no messages from afar skymds03 calls `get_next_local_mess()`
- prepare and clean out a pager record. As this may contain one of a range of message types, the largest size possible must be set-up.
- Check the number of unsent messages.
 - If there are more than 50, and this is an alarm, then don't send it. There will already be alarms, and if we add an alarm every time the number of messages is greater than 50 – which it already is – the queue will become packed with alarms
 - If there are more than 100 unsent messages don't send and "info pager" messages
- call the function pointed to by the `(mess_to_encoder*)()` function pointer. It may be any one of the functions in the following table:

Description	function called	Internal code
send a dummy / test message	<code>mess_to_dummy()</code>	DUMMY_TYPE
send a fax	<code>mess_to_fax()</code>	FAX_TYPE
send a email	<code>mess_to_email()</code>	EMAIL_TYPE
MERP format pager message	<code>mess_to_merp()</code>	MERP_TYPE
PET message forwarded to Telecom New Zealand	<code>mess_to_pacnet()</code>	NZ_X25_TYPE
Set up a reminder	<code>mess_to_reminders()</code>	REMINDER_TYPE
Send a SMS message to a SMSC	<code>mess_to_smpp_server()</code>	SMPP_TYPE
Send a message to the equatorial satellite	<code>mess_to_equatorial()</code>	SATELLITE_TYPE
A pager message to the encoder (for broadcast)	<code>mess_to_xacom()</code>	XACOM_TYPE
Send a PET protocol message	<code>mess_to_pet()</code>	PET_TYPE
Send a MTP format message	<code>mess_to_mtp_serv()</code>	MTP_TYPE
Send a reply to an email.	<code>mess_to_email_reply()</code>	EM_REPLY_TYPE
Forward a message to another <System_Name> site	<code>mess_to_remote_03()</code>	LAN_TYPE

Table 172: skymds03: message handling functions

- if we used the last option, we send a final message to the remote <System_Name> installation
- now act based on the success / failure status encountered when calling the function as per above

Internal Code	Action
SUCCESSFUL.	All OK. Update the message queue and history.
SUCCESS_NOHIST_UPD.	Update the message queue only, though for GSM type still update the history with the message id returned from the SMSC indicating which carrier used.
SUCCESS_AND_PAUSE.	Update the message queue and history file, and then pause for 30 seconds
ERR_NO_RETRY.	Update message queue only. (failure: don't retry)
ERR_RETRY.	Pauses for a while, and then re connect to network and try sending the message again
SMS_SUCCESSFUL.	Update the message queue and history file (SMS)
ERR_RECONNECT.	Re connect to the network and send the message again
ERR_REINIT_CONNECT.	Re initialize the connection and then re connect to the network
EXIT.	exit the skymds03.

Table 173: skymds03: message delivery response actions

- Finally, if we are not using our first-choice network connection, now is the time to change.

28.5.2 FUNCTION INITIALISE

initialise() prepares skymds03's run-time environment

- read and store the required environment variables
- parse the command-line
- catch a few signals
- check that the destination queue passed on the command-line is real, i.e. it appears in the destinations database table
- do a few bits of other database stuff.
- setup the shut-down semaphore. This is tested to determine when skymds03 should stop.
- if this instantiation of skymds03 will be receiving messages from another <System_Name> site call **init_local_03()**. Possibly should be **init_remote_03()??**
- call **HIST_start_session()** to prepare for writing to the history files.
- open all the database tables we are going to need
- call **init_network_connection()** to, well, initialise the network connection.
- open and prepare the maintenance queue
- attach to the public holiday shared memory.

28.5.3 FUNCTION: INIT_NETWORK_CONNECTION

init_network_connection() ensures a connection to one of the <System_Name> networks, and then, based on what kind of messages we will be sending, assigns the correct message processing function. It also does a few other bits and pieces based on the network connection / message protocol type.

- call **init_connection()** to connect to the TCP/IP LAN, the X.25 network, or a copy of skymds03 at some other <System_Name> site.
- if **init_connection()** failed, read the next record in the network connection table and try the step above again

This continues until a connection is made.

- Now, based on the ‘protocol’ field of the network connection record used, assign the (mess_to_encoder*) function pointer to the correct message processing function, and call the appropriate message-type specific initialisation function

Internal code	Processing Function	Initialisation Function
DUMMY_TYPE	<code>mess_to_dummy()</code>	<code>== none ==</code>
FAX_TYPE	<code>mess_to_fax()</code>	<code>init_fax()</code>
EMAIL_TYPE	<code>mess_to_email()</code>	<code>init_email()</code>
MERP_TYPE	<code>mess_to_merp()</code>	<code>init_merp()</code>
NZ_X25_TYPE	<code>mess_to_pacnet()</code>	<code>init_pacnet()</code>
REMINDER_TYPE	<code>mess_to_reminders()</code>	<code>init_reminders()</code>
SMPP_TYPE	<code>mess_to_smpp_server()</code>	<code>init_smpp_server()</code>
SATELLITE_TYPE	<code>mess_to_equatorial()</code>	<code>init_equatorial()</code>
XACOM_TYPE	<code>mess_to_xacom()</code>	<code>init_xacom()</code>
PET_TYPE	<code>mess_to_pet()</code>	<code>init_pet()</code>
LAN_TYPE	<code>mess_to_remote_03()</code>	<code>== none ==</code>
MTP_TYPE	<code>mess_to_mtp_serv()</code>	<code>init_mtp_serv()</code>

Table 174: skymds03: message handling functions

- finally a few details based on the network connection type are written to a port record and stored with `PORT_update_port_rec()`

29 skymds03 sub-modules

29.1.1 OVERVIEW

As mentioned before, skymds03 uses more than 40 C modules that define external functions. These modules are compiled separately, and <Company_Name>ed together with the main program.

This section describes only 11 major modules that provide the program with the major functionality. The following is a list of the major modules, along with its functionality and the functions through which the functionality is performed:

Module	Description
mdssende.c.	Email functionality.
mdssendf.c	fax functionality.
mdssendp.c	functionality for pager entry terminal (PET) protocol connection.
mdssendm.c.	functionality for message entry and retrieval protocol (MERP) connection.
mdssenda.c.	message delivery to the message transaction protocol (MTP) server.
mdssendw.c	message delivery to XACOM encoder with a single character return.
mdssendx.c	message delivery to XACOM encoder with reply.
mdssendz.c	functions for message delivery among short message service centres (SMSC) via “Message Peer to Peer Protocol” (SMPP).
mdssendv.c.	functions for message delivery from tone and voice to reminders.
mdssendn.c	Delivery to Telecom New Zealand ⁴⁰ via PACNET X.25.
mdssendd.c.	A dummy module that displays a message if run in debug mode.

Table 175: skymds03: message handling functions

29.2 MODULE: MDSSENDE.C: SEND EMAILS

The functions perform skymds03’s email functionality.

29.2.1 FUNCTION: INIT_EMAIL

This function performs initialization for email delivery:

- Get the environmental variables
- Open the line file
- Opening the email master file for checking overlap report.

29.2.2 FUNCTION: FINALISE_EMAIL

finalise_email() closes the line file and email file opened during the initialization. and is called when the email delivery process is over.

⁴⁰ Telecom is New Zealand’s (Telstra-like) dominant Teleco service deliverer and monopoly hardware owner.

29.2.3 FUNCTION: MESS_TO_EMAIL

mess_to_email() sends a message to email server.

- Load all the global variables to be used
 - If it is email report:
 - ~ Load the message into the email parameter structure
 - ~ Set record type to be email report type
 - ~ Set email type appropriately
 - If it is email message,
 - ~ Set record type to be email message type
 - ~ Set email type to be one of the following
 - ~ Extract the email subject from the message
 - Read the Email record to extract the required details including following record type, id type, pager contact id, line number, and sequence number
 - Now load the rest of the Global Variables from the Email file including email file name, email address
 - construct an email message and store it appropriately based on the email type.
 - use the unix script "/opt/mds/email/bin/send_to_email" which uses the unix email utility "/usr/bin/mailx"
-

29.3 MODULE: MDSSENDF.C: SEND FAXES

mdssendf.c contains the fax utility functions, used for both batch and 'on-line' fax jobs.

29.3.1 FUNCTION: INIT_FAX

init_fax(), as the name suggests, performs initialisation for fax delivery,

- Read the required environmental variables
- Open the Client stats master, Pager stats master, Fax master, and Delivery stats master database tables

29.3.2 FUNCTION: MESS_TO_FAX

This function sends a message to fax server. It requires three parameters, that is, pager number, message length, and the message to be sent. In particular, it does the task in several steps:

- Load all the global variables now to be used throughout the program
- If it is fax report, the record type is set to 'fax report' type and 'fax type' is set to one of
 - ~ FAX_LINE_REPORT,
 - ~ FAX_PAGER_REPORT,
 - ~ FAX_STARTEL_REPORT,
 - ~ FAX_WORK_REQ_REPORT,
 - ~ FAX_FULL_FORM,
- If a fax message the record type is set to 'fax message' and 'fax type' is set to one of
 - ~ FAX_INDIAL_DEST,
 - ~ FAX_PAGER_DEST
- Read the Fax record to extract the required details, which include record type, id type, pager contact id, and sequence number.
- Get the fax number
- Construct the fax message(s) and put them into files based on one of the following types:
- Use one of the unix scripts in the directory "/opt/mds/trufax/bin" to copy the fax files to fax server.

29.4 MODULE: MDSSENDP.C: SEND PET MESSAGES

The functions defined in mdssendp.c are PET protocol functions, all invoked by the program skymds03.

29.4.1 FUNCTION: INIT_PET

`init_pet()` performs the initialisation for each pager entry terminal connection,

- Open a communication port for the PET connection
- If it is the dialup connection, with or without timeout, then get environmental variable MDS_MODEM_INIT_STR and set modem_init_str to be that value.
- Get environmental variable MDS_MODEM_CONNECT_REPLY and set modem_connect_reply to be the value.
- Set modem on hook flag and logon pet flag to be false.

29.4.2 FUNCTION: FINALISE_PET

This function checks to see what log off process we have to do if there is any requirement. The log off process is always needed for the dial up PET, but for the leased line, log off process is needed only when the program is being shut down.

- If it is the dial up connection without timeout, or leased line connection, then return without doing anything.
- If it is logged on for the pet connection, then log off from a PET device, by doing the following
 - ~ Send an EOT character followed by a carriage return (ASCII 004 and 015) to the communication port
 - ~ Clear the input buffer
 - ~ Set logon pet flag to be false
- If modem is on hook, then disconnect from modem, by doing the following
 - ~ If it is dialup connection, with or without timeout, then send string “+++\r” to communication port, sleep for 1 minute and then send string “ATrATH\r” to the port
 - ~ Close the communication port
 - ~ Set modem on hook flag to be false

29.4.3 FUNCTION: MESS_TO_PET

`mess_to_pet()` sends a message to a remote PET receiver.

- If only a tone can be heard set the message length to zero
- Set up the buffer to hold the message that is going to be sent to PET device
- Put the message into the buffer and set up the following information in the buffer
 - ~ date/time stamp if appropriate
 - ~ pager id display if appropriate
 - ~ source display if appropriate
 - ~ sequence number if appropriate
 - ~ and finally check sum to the message buffer
- Truncate the message if it is too long, and then send the message in the buffer to the PET device

29.4.4 FUNCTION: SEND_MESSAGE

`send_message()` actually sends the message to the PET device we have dialled into. It requires one parameter, pager number.

- Send the pager number and message to the PET device
- Wait for up to five seconds for the PET device to reply
- If the response is
 - ~ ACK (ASCII 0x06): the message has been successfully sent.
 - ~ NAK (ASCII 0x15): then retry to send the message.
 - ~ RS and CR (ASCII 0x1e and 0x0d): the message is delivered, but with some error.
- Log a message into the history file to indicate the delivery result: failure, successful, or error.

29.4.5 FUNCTION: LOGON_PET

`logon_pet()` logs us on to a PET device..

- Repeatedly do the following until successfully logged on or the maximum number of retries reached.
 - ~ repeatedly try to get a correct logon prompt from the PET device
 - ~ if not get correct logon prompt for a certain number of retries, exit the function with error
 - ~ send a carriage return to the PET device
 - ~ wait up to five seconds to receive the ID= prompt from PET device
 - ~ tell the PET device that the program whishes to operate in remote mode
 - ~ wait up to three seconds for the PET device to reply
 - ~ if the response is ack, logon successful
- If the number of retries exceeds a certain number of times, then exit the program with error
- If the logon is successful, then set the logon pet flag to be true.

29.4.6 FUNCTION: DISCONNECT_MODEM

This function drops DTR into the modem so that the modem disconnects.

- If it is a dialup connection, with or without timeout limit, then send a string “+++\\r” to the communication port.
 - Sleep for 1 minute, and then send another string “AT\\rATH\\r” to the communication port.
 - Sleep for another 1 minute as modem is a slowly response device
 - Close the communication port
 - Set modem on hook flag to be false.
-

29.5 MODULE MDSSENDM.C: SEND MERP MESSAGES

This module defines functions that are all MERP protocol related, and are invoked by the program skymds03.

29.5.1 FUNCTION: INIT_MERP

`init_merp()` performs the initialization for a MERP (message entry and retrieval) connection.

- Open a MERP communication port
- Send a message with sequence number of 00 to MERP device
- Wait for the response from the MERP device. If the response is ok, then return with success. Otherwise re send a message with sequence number of 00 to MERP device until the response is successful, or error received.
- Set the current sequence number to be 01

29.5.2 FUNCTION: MESS_TO_MERP

This function sends a message to a MERP receiver. It requires three parameters, that is, pager number, message length, and the message to be sent. In particular, it does the task in several steps:

- If it is a tone only with answering, then set the message length to be zero such that the message becomes null.
- Set up the message with the date/time stamp, pager id, the message source, and a sequence number
- Then three times:
 - ~ Send the message to MERP device
 - ~ Wait for the response from MERP device
 - ~ If the response is error, exit the program with error.
- If the message was sent successfully, return with success.

29.5.3 FUNCTION: RESET_SEQU_NBR

This function sends a message with sequence number of 00 to reset the remote system. It performs the task in following steps:

- Set up the message buffer with a sequence number of 00
- Send the message to remote MERP device
- Wait for the response from the remote MERP device. If the response is ok, then return with success. Otherwise try to re send the message again and again until the ok response received.
- Set the current sequence number to be 01 now.

29.6 MODULE MDSSENDA.C: SEND MTP MESSAGES

The functions defined in the module are (MTP) message transaction protocol related, and are invoked by the program skymds03. There are only two functions for skymds03 to use.

29.6.1 FUNCTION: INIT_MTP_SERV

This function initializes the MTP server and connects to the MTP server. It uses the function cvrt_int to get IP number and the function GEN_connect_server to connect to MTP server.

- Initialize the MTP serial number to be 00
- Get IP port number from the host address via invoking the function cvrt_int.
- Try to connect to MTP server forever until successful, via the function `GEN_connect_server()`
- Sleep for 5 seconds and try to connect to MTP server again if not successfully connected
- Return with success if successfully connected

29.6.2 FUNCTION: MESS_TO_MTP_SERV

This function sends a message to MTP server that has been connected via using the function `init_mtp_serv()`.

- Get the transaction type from the message to be sent, via using function cvrt_init.
- Get the field delimiter from the message to be sent.
- Pack the message and put the packed message into a buffer
- Send the packed message to MTP server via invoking the function GEN_send_to_server
- Try to obtain the serial number from the MTP server. If a connection can not be made to the MTP server, or the serial number is incorrect
 - ~ Close the communication port
 - ~ Sleep for 2 seconds
 - ~ Connect to MTP server again with `init_mtp_serv()`
 - ~ Return with error retry
- Get the transaction type from the response packet, and compare it with the original one and return with error if they do not match

29.7 MODULE MDSSENDW.C: MESSAGES TO AN XACOM ENCODER

The functions defined in this module contain functions that initialize the XACOM encoder with single character reply and deliver messages to the encoders.

29.7.1 FUNCTION: INIT_EQATORIAL

`init_equatorial()` reads the records in the destination file. It ignores the group destination records and non-satellite destination records.

- Open the destination file separately from the main program so that all destinations can be read in to get the satellite encoder addresses
- Set up to use the second index with `set_cisam_index()`
- Read the record one by one, in the destination file.
- If the record is a group destination record or a non-satellite destination record, then ignore it. Otherwise put it into an array structure.
- Close the destination file

29.7.2 FUNCTION: MESS_TO_EQATORIAL

`mess_to_equatorial()` sends a message to a xacom encoder and waits for the reply.

- If it is a tone only with answering, then set the message length to be zero.
 - Set up the buffer to hold the message and pick up the actual encoder address.
 - Pack up the message to be sent, date/time stamp, pager id, source, and sequence number
 - Truncate the message if it is too long
 - Send the packed message to the encoder
 - Clear up the input buffer and wait up to 5 seconds for the XACOM encoder to reply.
 - OK: return with success.
 - Error: ring the bell seven times and return with error.
-

29.8 MODULE MDSSENDX.C: MESSAGES TO XACOM ENCODERS (WITH REPLY)

mdssendx.c contains functions for message delivery to XACOM encoder with reply.

29.8.1 FUNCTION: INIT_XACOM

init_xacom() reads the destination file and sets up channel and time offset for each of the satellite encoder addresses.

- Open the destination file separately from the main program so that all destinations can be read in to get the satellite encoder addresses
- Open the destination file
- Read all records one by one, and ignore all records that are not satellite encoder addresses.
- Set up channel and time offset for each destination
- Close the destination file

29.8.2 FUNCTION: MESS_TO_XACOM

mess_to_xacom() sends a message to a xacom encoder and waits for the reply.

- If it is a tone only with answering, then set the message length to be zero.
- Set up the buffer to hold the message and pick up the actual encoder address.
- Pack up the message to be sent, date/time stamp, pager id, source, and sequence number
- Truncate the message if it is too long
- Send the packed message to the encoder
- Clear up the input buffer and wait up to 5 seconds for the XACOM encoder to reply.
- OK: return with success.
- Error: ring the bell twelve times and return with error.

29.9 MODULE MDSENDL.C: SEND MESSAGES TO OTHER <SYSTEM_NAME> SITES

mdssendl.c contains functions for message delivery via LAN to another skymds03 process.

29.9.1 FUNCTION: INIT_REMOTE_03

init_remote_03() connects, via the LAN, to a remote socket, established by an instance of skymds03 running at some other <System_Name> site.

- Get the remote host port number via using the function cvrt_init
- Clear up the socket address structure
- Get the remote host IP address by referring to the host file /etc/hosts. If the item for the remote host is not contained in the file, then an error message is generated.
- Construct the socket address structure
- Open a socket
- Bind the remote address structure to the socket
- Set the serial number to be 00.

29.9.2 FUNCTION: INIT_LOCAL_03

`init_local_03()` prepares the local end of a socket to socket connection.

- Clear up the socket address structure and set up the address structure for a socket
- Create a socket
- Bind the server's address to the socket. If it fails, then an error message is generated and return with error.
- If all successful, then return with success.

29.9.3 FUNCTION: MESS_TO_REMOTE_03

`mess_to_remote()` sends a message to a skymds03 process at another <System_Name> site.

- First Check the queue header to see if it is valid.
 - Clear the system error number variable. This will be checked if there is a problem later.
 - Fill the message packet to be sent.
 - Send the message to remote skymds03 process via network socket
 - Check the right response from the remote skymds03 process. If not then exit with an error.
 - If the response is received increase the sequence number by 1. Success.
-

29.10 MODULE MDSSENDV.C: ADD REMINDERS

mdssemdv.c contains functions that send reminders (via the Reminders Shared Memory) to the Call Centre Operators

29.10.1 FUNCTION: INIT_REMINDERS

- Open the temporary message and the reminders database tables
- Attach to the reminder and global shared memory segments

29.10.2 FUNCTION: MESS_TO_REMINDERS

This function sends a message to reminders.:

- Get the temporary message record by invoking the function `get_temporary_message` with parameters of pager number, message length, and message
- Open the line master file, read the file for the temporary message record, and then close the file.
- If it fails to find the temporary message record in the file, then return with error.
- If the record was found in the master line file, then add it into the reminder shared memory and return with success.

29.11 MODULE MDSSENDZ.C: SENDING SMS MESSAGES

This module contains functions for message delivery among short message service centres using the Short Message Peer to Peer protocol (SMPP).

29.11.1 FUNCTION: MESS_TO_SMPP_SERVER

This function sends a message to skymds11 process that runs as SMPP server.

- If the length of the message to be sent is 0, then log and skip the message as SMSCs can not handle any message with 0 length.
- Prepare the message: set up date/time stamp, sequence number, check the length and possibly truncate, remove the prefix, and set the field lengths
- Allocate a piece of memory as a buffer and place the packed message into the buffer
- Save a serial number for later comparison with the one that comes from the server's response
- Check if the mobile number is valid.
- Search for destination country code stored in the encoded address
- Send the packed message to the SMPP server.
- ... wait ...
- Read the response from SMPP server
- Check the response. Don't retry, just return with the error (or success) status.

29.11.2 FUNCTION: UPDATE_MSG_ID_AND_DELIV_STATUS

This function updates the primary history record with the message id returned from the SMSC.

- Set up the new history primary record that contains multiple fields, which include pager id type, pager contact id, date received, time received, line number, and machine id.
- Update the primary history record via invoking the function HIST_update_prim.
- Check if the current dispatch centre is identical to the dispatch centre of our city. If it is, then set the management flag to be local. Otherwise set it to be remote.
- Set up the new secondary record to update. The record contains field transaction type, date finalized, time finalized, SMSC id, and delivery status
- Try to update the secondary history record. If the record is not found in the secondary history file, add the secondary record into the secondary record file.
- Check if the current dispatch centre is identical to the dispatch centre of our city. If it is, then set the management flag to be local. Otherwise set it to be remote.

29.11.3 FUNCTION: CHECK_MOB_NUM

This function checks if the given mobile number is valid.

A valid mobile number contains at least 8 characters and a (varying) maximum number of characters specified by other details concerning the mobile. It must not contain any space or non-numeric character.

29.12 MODULE MDSSENDN.C: MESSAGES TO TELECOM NEW ZEALAND

mdssendn.c contains functions for message delivery to Telecom New Zealand over the PACNET X.25 network. As noted before, Telecom is the New Zealand version of Telstra, the massively dominant telecommunications service provider and monopoly network owner.

29.12.1 FUNCTION: INIT_PACNET

init_pacnet just sets the `pacnet_initial_connection` flag to be true.

29.12.2 FUNCTION: MESS_TO_PACNET

This function sends a message to Telecom New Zealand via X.25 PACNET. A new connection being made for each message sent.

- Check if the initialization was previously called. If not, re-connect to PACNET.
- Discard all characters up to and including the trailing <CR><LF>, and then receive ID= prompt
- Send the Telecom New Zealand pager or phone ID and the message in the format:

<page/phone ID>@<message><CR>

- Get the response code for the message sent above, in the format:

?<2-digit code> <Response Message><CR><LF>

- Check the response code and look for the start of the response (begins with a '?').
- If message OK disconnect the X.25 connection, else log an error and return.

29.13 MODULE MDSENDD.C – SEND TEST / DUMMY MESSAGES

This module contains one function `mess_to_dummy()`, which is used by skymds03 process to send a message to a dummy xacom encoder. This function does nothing but displaying the pager number, message length, and message when the program runs in the debug mode.

30 skymds25

30.1 SKYMDS25 OVERVIEW

skymds25 sends messages and updates from the <System_Name> system to <Company_Name>'s SKY<COMPANY_NAME> billing system. Compared to the extensive messaging capabilities built in skymds25's partner program skymds18, skymds25 itself only sends:

- temporary messages
- SMS phone addition, deletions, and changes
- pager holder name changes
- answer phrase changes
- follow-me updates (messages being broadcast in an alternative area)
- SAGRN (Sth Australian Government) error code changes

to the Wang. Each of these is handled by a different function, the remainder of skymds25 consisting of support utilities.

30.2 RUNNING SKYMDS25

skymds25 is run from the start.mds start-up script. The supervisor / administrator running the start-up script is prompted with:

```
Start skymds11, skymds67, skymds18, skymds25, skymds28, skymds81,  
skymds06, skymds27, skymds10, remterm and skymds20 (y/n) :
```

and, presuming they answer 'y', skymds25 is run.

30.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-n < pager number >	Only send transactions for this pager to the Wang
-l	No Comms: don't connect to a communications port
-e < seconds >	Timeout. (Default = 30 seconds)
-t < comms port >	The port that skymds25 communicates (sends messages) via
-D	run in debug mode, listing detailed run-time information.

Table 176: skymds25: Command Line Parameters

30.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute skymds25 at one <System_Name> site is:

```
$MDS_BIN_DIR/skymds25 -t /dev/x25010 2>$LOG_SCR
```

indicating that skymds25 is run:

- sending messages to the port /dev/x25010
- a default 30-second timeout is used – no '-e' parameter used.
- the stderr is copied to the device LOG_SCR, defined in start.env to be /dev/tt0p5, a terminal somewhere

30.3 PROGRAM DESCRIPTION

As noted, skymds25 sends messages from a <System_Name> system to <Company_Name>'s Wang. skymds25's main function runs as a loop, repeatedly calling `get_next_mess()`, then `process_transaction()` to process the message. `process_transaction()` calls one of 6 handler functions to process the 6 types of updates going to the Wang:

- temporary messages
- SMS phone addition, deletions, and changes
- pager holder name changes
- answer phrase changes
- follow-me updates (messages being broadcast in an alternative area)
- SAGRN (Sth Australian Government) error code changes

skymds25 reads the *maintenance* queue for messages (from all over Australia), these (maintenance) queues functioning as its distribution queues.

30.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds25.

30.4.1 SOURCES FILES

Source File	Description
skymds25.c	Interface to Wang stock and billing computer to send information to the Wang
crccalc.c	Calculate the CRC for a given string.
gensubs.c	Contains various general-purpose routines that are separately compiled.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
mdscomio.c	Various serial io routines
mdsginit.c	A number of functions that perform commonly used initialisation code segments.
mdslog.c	Generic error logging.
mdslread.c	Line file read functions during Austel Number Change.
mdsportn.c	Routines associated with the ports file.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdssemcck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions that perform operations to do with time.

Table 177: skymds25: Source Files

30.4.2 HEADER FILES

Header File	Description
skymds25.h	Wang (SKY<COMPANY_NAME>) transaction packet formats
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.: C time field definitions
macros.h	User defined macros.: definitions & macros for common functions
mdscdr.h	CDR definitions file
mdstype.h	Used for validating message chars.
mdsderec.h	Ports security master file
mdserrec.h	GL3000 definition file: GL3000 error file
mdsfarec.h	Pager facilities master file
mdsflsz.h	Standard <System_Name> includes.: field length & definitions
mdsgl3000.h	GL3000 definition file: GL3000 layout
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file

Table 178: skymds25: Header Files

Header File	Description
mdsparam.h	parameters used by PMS system
mdsparec.h	Pager Master File
mdsporec.h	Ports security master file: Ports Record
mdsporec.h	Ports security master file: Ports Record
mdspsec.h	Password master file
mdsquarec.h	Contacts Message queue file: Destination Message queue file
mdstmrec.h	Temporary message master file
mdstrrec.h	MNTQ: maintenance queue file
mdswarec.h	Wang interface comms layout

Table 179: skymds25: Header Files (continued)

30.4.3 DATABASE TABLES

Each of these header files contains (only) a C-struct that matched the structure of a database table used by skymds25.

Database Table	Header File	Description
mdsdestm	mdsderec.h	Destinations master
mdserror	mdserrec.h	generic error
mdsfacil	mdsfarec.h	Facilities – what special <System_Name> services can be used
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdspassw	mdspsec.h	Password username
mdsports	mdsporec.h	Ports security
mdsports	mdsporec.h	Ports security
mdsqal.txt	mdstrrec.h	Inter processor transaction queue
mdsqxx.txt	mdsquarec.h	Destination queues
mdstmesg	mdstmrec.h	Temporary message master

Table 180: skymds25: Database Tables

30.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description	Example
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide ...	O
MDS_DATA_DIR	The root of the data directories, queues, database and history.	/var/mds/data

Table 181: skymds25: Environment Variables

30.5 SKYMD25: MAJOR FUNCTIONS

30.5.1 FUNCTION: MAIN

- call **initialise()** to prepare skymds25's runtime environment

Now the infinite loop, sending messages to the Wang, is started

- Test the shutdown semaphore. If we want to shut down we do it at the start of each loop.
- Call **get_next_mess()** to obtain the next message that needs to be sent to the Wang
- Call **process_transaction()** to send the message to the Wang
- Call **update_queue()** to update the queue header.

30.5.2 FUNCTION: INITIALISE

The **initialise()** function sets up skymds25's run-time environment, via the following steps:

- Read the environment variables MDS_MACHINE_ID and MDS_DATA_DIR.
- parse the command-line
- call **background()** to run skymds25 as a background process
- catch a few signals
- record in the mdsparts table that skymds25 has started, and open the communication port as a file.

If the letter 'l' was passed as a command-line parameter no connection to the communications port is made. Just send the messages to nowhere?

- change the permissions on the port. Remove the O_NDELAY flag, so that reads can take any amount of time, as opposed to faulting out on the first failure.
- lock the port, and build up a file parameter structure to set the port (file) up as we wish
- obtain and check the shutdown semaphore
- get the city code and the code of the queue that skymds25 is going to read from.
- open the maintenance queue and the distribution queue that skymds25 is going to read messages from to send to the Wang.

30.5.3 FUNCTION: GET_NEXT_MESS

get_next_mess() reads messages from the maintenance queue, messages that need to be replicated to the Wang.

- lock the maintenance queue and read the queue's header. Then unlock the queue
- read the record pointed to by the nextsend pointer in the maintenance queue's header.

Maintenance queue messages can be of varying lengths. Thus there are two reads, one to get the standard part of the maintenance queue message – which includes the 'message length' – and then the remaining 'message length' bytes are also read.

30.5.4 FUNCTION: PROCESS_TRANSACTION

process_transaction() is a switchboard function that, based on the message's file_id, calls the appropriate function to post the maintenance queue message to SKY<COMPANY_NAME>.

- first, if this update came from the Wang (SKY<COMPANY_NAME>) in the first place, then skip it.
- Next, as per the following table, decide which handler function is appropriate.

Action	file_id	sub_type	subscriber_type	Function Called
temporary messages	ID_MDSTMESG	(any)	(any)	process_trip_msg_trans()
SMS phone adjustments		' '	'S'	process_sms_phone_trans()
pager holder name changes	ID_MDSPAGER	(any)	'2'	process_holder_name_trans()
answer phrase changes	ID_MDSLNE	'1'	(any)	process_answer_phrase_trans()
Sth Australian Government error code changes	ID_MDSERROR	(any)	(any)	process_g13000_error()

Facilities.Follow-me.rec_type

follow-me updates	ID_MDSFACIL	FA_TYPE_AWAY	process_follow_me_trans()
-------------------	-------------	--------------	-----------------------------------

Table 182: skymds25: Maintenance Queue Message Handlers

Constant	Value	in file:
ID_SUB_PAGER_ALL	' '	mdsparam.h
SUBSCRIBER_TYPE_SMS	'S'	mdsparam.h
ID_SUB_PAGER HOLDER_NAME	'2'	mdsparam.h
ID_MDSTMESG	58	mdsparam.h
ID_MDSPAGER	7	mdsparam.h
ID_MDSLINE	4	mdsparam.h
ID_MDSFACIL	63	mdsparam.h
ID_MDSERROR	9	mdsparam.h
ID_SUB_LINE_ANSWER_PHRASE	'1'	mdsparam.h
FA_TYPE_AWAY	'F'	mdsfarec.h

Table 183: skymds25: Important Flags and Constants

- the return value from the sub function (`process_xxx()`) is returned from process_transaction as a return value.

30.5.5 FUNCTION: PROCESS_TMP_MSG_TRANS

`process_tmp_msg_trans()` updates the change in a (pager) temporary message to SKY<COMPANY_NAME>. A temporary message is information that the Call Centre Operator reads out to a caller when they first call a <System_Name> call centre.

- check that the transaction message we just obtained from the maintenance queue was from a pager.
- if a ‘-n pager number’ was passed on the command-line, and the current pager number is not for that pager, then exit
- The maintenance queue transaction type is updated:

from <System_Name> tran_type:	to SKY<COMPANY_NAM E> tran_type:		
A	01A		add
D	01D		delete
C	01C	error – carry on	change
U	01C	error – carry on	update

Table 184: skymds25: <System_Name> to SKY<COMPANY_NAME> trans_type mappings

- The start and end time of the temporary message are converted into the Wang format, and the temporary message text is copied to a Wang message
- `send_transaction()` sends the transaction to the Wang

30.5.6 FUNCTION: PROCESS_SMS_PHONE_TRANS

Sends various data about the state of SMS phones to the Wang.

- if a ‘-n pager number’ was passed on the command-line, and the current pager number⁴¹ is not for that pager, then exit
- The messages are handled as per:

Add: <System_Name> Trans Type A

- The SKY<COMPANY_NAME> / Wang `trans_type` is set to 07A
- The phone status is set to the constant PHONE_ACTIVE_STATUS (= ‘A’)

⁴¹ sms telephones are treated as pagers that receive SMS messages.

Change: <System_Name> Trans Type C

Same as 'Update: Trans Type U'

- ~ If the phone ('pager') is flagged as Active, Suspended, or Inactive, this is copied to the Wang transaction record
- ~ The **trans_type** is set to 07C (Active, suspended) or 07D (Delete)

Update: <System_Name> Trans Type U

Same as 'Change: Trans Type C'

- ~ If the phone ('pager') is flagged as Active, Suspended, or Inactive, this is copied to the Wang transaction record
- ~ The **trans_type** is set to 07C (Active, suspended) or 07D (Delete)

Delete: <System_Name> Trans Type D

- ~ The phone status is set to PHONE_CANCELLED_STATUS on the Wang Transaction record
- ~ The **trans_type** is set to 07D (Delete)

- Other necessary information is copied to the Wang transaction record
- **send_transaction()** is used to send the transaction to SKY<COMPANY_NAME> on the Wang.

30.5.7 FUNCTION: PROCESS HOLDER NAME TRANS

process_holder_name_trans(), as the name suggests, updates the name of the person (the pager holder) attached to the pager record.

- check that the size of the maintenance queue record is larger than the size of the holder's name. Later it is tested to be exactly twice the size. Whatever.
- if the '-n pager number' option was passed in on the command line and the current pager number does not match that pager number, then exit.
- Add, Delete (<System_Name> trans_types A and D) are errors also. Exit.

Change: <System_Name> trans_type C

- ~ set the Wang trans_type to 02C
- ~ copy the pager holder's name from the maintenance queue message

Update: <System_Name> trans_type U

- ~ set the Wang trans_type to 02C
- ~ copy the pager holder's name from the maintenance queue message

- copy a couple of other details to the Wang record
- Call **send_transaction()** to send it to the SKY<COMPANY_NAME> / the Wang

30.5.8 FUNCTION: PROCESS ANSWER PHRASE TRANS

Sends an updated answer-phrase for recording in the Wang. Additions, deletions and changes are errors, though changes will still be implemented as updates.

- Add: <System_Name> Trans Type A
- Change: <System_Name> Trans Type C Error
- Delete: <System_Name> Trans Type D

- Update (Change): Set the SKY<COMPANY_NAME> trans_type to be 05C, and copy the answer phrase to the Wang.
- Use **send_transaction()** to send the answer-phrase containing message to the Wang.

30.5.9 FUNCTION: PROCESS_GL3000_ERROR

`process_gl3000_error()` adds a new GL3000 error code to those already stored in SKY<COMPANY_NAME>. GL3000 is a <System_Name>-connected system used by the South Australian government.

- if trans_type is A, then set the Wang trans_type to be 08A. If it is not ‘A’ we have an error.
- fill up a Wang / SKY<COMPANY_NAME> transaction record with some further info.
- call `send_transaction()` to post the transaction to the Wang.

30.5.10 FUNCTION: FOLLOW_ME_TRANS

Follow-Me is the <Company_Name> service that broadcasts messages to a pager in an alternative city or region for a set period. This period matches the time that the pager holder (and pager) will be in this alternative area, so that they can still receive their messages.

Add: <System_Name> Trans Type A

- ~ The SKY<COMPANY_NAME> / Wang `trans_type` is set to 03A
- ~ fill up a Wang / SKY<COMPANY_NAME> transaction record with some further info. Use the utility functions `want_last_op()` and `wang_dest()` to do some of the translations
- ~ call `send_transaction()` to post the transaction record to the Wang.

Delete: <System_Name> Trans Type D

- ~ The SKY<COMPANY_NAME> / Wang `trans_type` is set to 03D
- ~ fill up a Wang / SKY<COMPANY_NAME> transaction record with some further info.
- ~ call `send_transaction()` to post the transaction record to the Wang.

Change: <System_Name> trans_type C

- ~ First, the same procedure as for ‘delete’ is followed. (type set to 03D)
- ~ If the return value from `send_transaction()` doesn’t indicate a problem, the steps as for ‘Add’ are used (type = 03A)

Update: <System_Name> Trans_type U

- ~ First, the same procedure as for ‘delete’ is followed. (type set to 03D)
- ~ If the return value from `send_transaction()` doesn’t indicate a problem, the steps as for ‘Add’ are used (type = 03A)

30.5.11 FUNCTION: SEND_TRANSACTION AND SEND_TRANS_REC

These two functions send prepared Wang / SKY<COMPANY_NAME> messages to the Wang, `send_transaction` being merely a wrapper for `send_trans_rec()` (plus a 1-9 sequence number).

30.5.11.1 Function: send_trans_rec

`send_trans_rec()` actually writes the transaction record to the Wang.

- truncate the record if it is too long.

Surprisingly the procedure doesn’t exit here (if the record is too long). A truncated record will still be written to SKY<COMPANY_NAME>.

- The transaction message is copied to the local Wang Record
- if the letter ‘l’ was passed on the command line – “No Comms” – then exit out returning TRUE, indicating success. “No Comms” means no actual write to SKY<COMPANY_NAME>
- In an infinite loop the record is written to the Wang until an ACK (acknowledge) message is received back.

- The first thing in the loop is testing the shutdown semaphore. If it is set then time to shut down!
- Next, if the retry_count is 3, `send_reset_sequ()` is used to reset the sequence count. The retry_count is then set to -1. Sequence numbers only vary between 0 and 9.
- The port – opened as a special block file – is flushed
- Now the Wang Record is written to the comms port (file). The port has been opened with the delay flag cleared, so that data can flow through at any rate. To compensate for this the `write()` call is wrapped in `alarm()` calls, so that an alarm will be raised if it takes more than `WRITE_TIMEOUT` (10) seconds for the write to be completed.
- Once the write is completed we read the port to find an ACK acknowledge signal. The timeout for reading is either passed in on the command-line when `skymds25` is run (the ‘-e’ parameter), or it defaults to `DEF_COMM_TIMEOUT` (30 seconds).
- The data read is searched backwards for an ACK (acknowledge) or NAK (negative acknowledge) signal. If neither is found the whole write / read is tried again.
- If an ACK was not found then the sequence number is reset to 0 and a cyclic redundancy Check (integrity check) is done on the received data and the whole write / read is tried again.
- If an NAK (negative acknowledge) was found the same thing is done (round again)
- Only get past these steps if an ACK has been read.
 - Wrong sequence number read in return: sleep for 5 seconds and go around again
 - correct sequence number: all done.

30.5.12 FUNCTION: UPDATE_QUEUE

`update_queue()` updates the maintenance queue header and details record once a record has been read from the queue.

- Lock the maintenance queue
- read and check the maintenance queue header
- advance the next-send index by one
- write the new header back to the queue.
- unlock the queue header

30.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

30.7 STOPPING SKYMD25

1. Set the `shutdown_semid` to non-zero.
2. Find its PID and KILL -9 it (as root)

31 skymds10

31.1 SKYMD10 OVERVIEW

skymds10 provides a TNPP (Telocator Network Paging Protocol) <Company_Name> between <System_Name> and an external site. At the time of writing this has been implemented for only the South Australian Government Radio Network (SAGRN) running out of Adelaide.

Most of the functionality (the tnpp-side logic) is coded in mdstnpp.c with a few functions (the <System_Name> side), back in skymds10.c. A third source file (mdstnsub.c) contains support and debugging routines.

31.2 RUNNING SKYMD10

skymds10 runs on the two Melbourne servers (lnkm and lnko) and the main Sydney server lnks

31.2.1 COMMAND LINE PARAMETERS

Parameter	Description
<u>Mandatory</u>	
-a <port number >	our (source) address in TNPP format.
-t < port >	The port against which skymds10 is registered.
<u>Optional</u>	
-D	Debug Mode, write out detailed information as skymds10 runs
-c	Transparent CRC mode [default = Hex]
-P	Identify pagers by their Pager number. [Default = by the pagers' unique 'Cap Code']
-f	Send pager messages in Flex format
-b <port number >	the destination address in TNPP format
-n < character >	Some type of destination identifier
-z < character >	The TNPP zone
-l < port >	The TCP/IP port on which skymds10 listens.
-s < seconds >	Sleep time, the time gap between message processing attempts ⁴² . [Default = 1 second]
-?	Call usage() to print an (limited) version of these command line parameters.

Table 185: skymds10: Command Line parameters

31.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute skymds10 at one <System_Name> site is:

```
$MDS_C_DIR/skymds10 -D -t /dev/vcg03 -P -a 3101 -b 6301 -c -1 3342 2>$MDS_LOG_DIR/tnpp.log
```

indicating that skymds10 is to be run:

- in debug mode
- registered in the mdsports table against the port /dev/vcg03
- identifying pagers by their pager number
- our TNPP address is 3101
- the destination (TNPP) address is 6301
- with Transparent (4 byte) CRC message integrity checking
- listening on TCP/IP port 3342
- errors, and as we are in debug mode, debug messages, are duplicated to the file MDS_LOG_DIR/tnpp.log

⁴² I don't think that this is actually used. The effective sleep time is hard coded as 10 (seconds) in mdstnpp.h.: #define TNRI 10

31.3 PROGRAM DESCRIPTION

skymds10 is the interface between a <System_Name> system and any other system using the TNPP protocol which, currently, is only the South Australian Government Radio Network. Control passes from the `main()` function to `tnpp_main()` which iteratively processes messages until the program terminates. `tnpp_main()`, coded in `mdstnpp.c`, repeatedly calls one of 5 functions depending on the current state of the process, each function call possibly changing the state of the system⁴³. These five functions themselves call a number of sub functions in order to accomplish the process of communicating successfully with the TNPP client.

Condition	Internal Constant	Value	Handler
(re) initialise the connection	TNPP_INIT	0	<code>tnpp_init_state()</code>
Wait for Remote communication	TNPP_AWAIT_ENQ	1	<code>tnpp_await_ENQ_state()</code>
Standard idle / poll state	TNPP_READY	2	<code>tnpp_ready_state()</code>
Transmit a message	TNPP_TX	3	<code>tnpp_transmit_state()</code>
Get a response from the remote site.	TNPP_AWAIT_TX_RESPONSE	4	<code>tnpp_await_tx_response_state()</code>

Table 186: skymds10: major behaviors and function calls

Each function resets the state, and `tnpp_main()` loop around.

As mentioned above, the <System_Name> side functions are coded in `skymds10.c` while the TNPP / SAGRN – side functionality appears in `mdstnpp.c`. While this second collection should become a library it does, currently, still have calls back to `skymds10.c`.

31.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into `skymds10`.

31.4.1 SOURCES FILES

Source File	Description
<code>form_path.c</code>	Make a file-path from an environment variable and a filename
<code>gensubs.c</code>	Contains various general-purpose routines that are separately compiled.
<code>isstart.c</code>	Wrapper for C-ISAM <code>isstart</code> function - choose index and position the read-point
<code>mdscdr.c</code>	Contains various general-purpose routines that are separately compiled.
<code>mdsemqmain.c</code>	Generic message email queuing routine.
<code>mdsginit.c</code>	A number of functions, which perform commonly, used initialisation code segments.
<code>mdsglsub.c</code>	GL3000 routines
<code>mdskbld.c</code>	Contains functions to build and unbuild pager keys, and set-up a dummy pager.
<code>mdslog.c</code>	Generic error logging.
<code>mdsportn.c</code>	Routines associated with the ports file.
<code>mdsqmain.c</code>	Generic message queuing routine.
<code>mdsqmnt.c</code>	Queues file updates for transmission to other machines.
<code>mdsqurtm.c</code>	Generic message queuing routine.
<code>mdsbusub.c</code>	generic functions for the message and data queues such as read/write, lock/unlock.
<code>mdssemcck.c</code>	Check the nominated semaphore and return TRUE if it has dropped.
<code>mdssetup.c</code>	Obtain set-up definitions from a text set-up file.
<code>mdsstamp.c</code>	Date time stamp & sequ nbr generation
<code>mdstmrtn.c</code>	Contains functions, which perform operations to do with time.
<code>mdstnpp.c</code>	TNPP Transceiver
<code>mdstnsub.c</code>	Support and debug display routines for TNPP transactions.
<code>skymds10.c</code>	Receive message from IP user datagram and deliver message to a TNPP device.
<code>spawn.c</code>	Spawns off a child process and (optionally) waits for its completion.
<code>tnppcrc.c</code>	Calculate the CRC for a given TNPP string.

Table 187: `skymds10`: Source Code files

⁴³ Yes all you computer scientists, `skymds10` is a finite state machine

31.4.2 HEADER FILES

Header File	Description
ascii.h	Function keys and special chars.: PMS function key values.
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.: C time field definitions
macros.h	User defined macros.: definitions & macros for common functions
mdsaprec.h	A-Party Master file
mdscdr.h	CDR definitions file
mdscdrec.h	CDR data structure
mdsderec.h	Ports security master file
mdsflsz.h	Standard <System_Name> includes.: field length & definitions
mdsg1rec.h	GL3000 tnpp record: <System_Name> -> GL3000 CDR record layout
mdsg2rec.h	GL3000 Billing record: GL3000 -> <System_Name> CDR record layout
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsparam.h	parameters used by PMS system
mdsparec.h	Pager Master File
mdsporec.h	Ports security master file: Ports Record
mdspssrec.h	Password master file
mdsqrec.h	Contacts Message queue file: Destination Message queue file
mdsrerec.h	Record reply back to remote host
mdssmpp.h	SMSC structure & size definitions.
mdstnpp.h	TNPP Constants and Codes (for SAGRN)
mdstrrec.h	MNTQ: maintenance queue file

Table 188: skymds10: Header Files

31.4.3 DATABASE TABLES

Each of these header files contains (only) a C-struct that matched the structure of a database table used by skymds10.

Database Table	Header File	Description
mdsaparty	mdsaprec.h	A-Party Customer Info
mdscdrec	mdscdrec.h	Call details master
mdsdestm	mdsderec.h	Destinations master
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsqrec.h	Destination queues

Table 189: skymds10: Database Tables

31.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description	Example
MDS_DATA_DIR	Where the <System_Name> data files reside	/opt/mds/data/
MDS_BIN_DIR	Where the <System_Name> executables are	/opt/mds/bin/
MDS_MACHINE_ID	What machine we are on S & X in Sydney, M & O Melbourne ...	O

Table 190: skymds10: Referenced Environment Variables

31.5 SKYMD10: MAJOR FUNCTIONS

31.5.1 FUNCTION: MAIN

Similar to many <System_Name> `main()` functions, first we call `initialise()` and then pass control to `tnpp_main()` to actually implement the central TNPP message transmission logic.

31.5.2 FUNCTION: INITIALISE

The initialise() function sets up skymds10's run-time environment, via the following steps:

- reads in the 3 needed environment variables.
- parse the command-line.
- use `background()` to run skymds10 as a daemon.
- catch virtually every possible signal.
- call `PORT_set_port_started()` to register skymds10 against the port passed in on the command line (via the '-t' parameter).
- attach to shared memory where every minute or so a time stamp is written which indicates to mdsalarm, the <System_Name> system monitoring process, that skymds10 is still running.
- read from the mdsparts database table the number of the shutdown semaphore for this process.
- call `tnpp_setup()` to initialise the communication stream with the remote TNPP system.
- open the database tables and the queues necessary.
- finally call `init_channel_zone()` which reads in the transmission frequency (channel and zone) information. As all current TNPP transmissions are to the SAGRN on frequency F6 (148.8125 MHz) this information is redundant

31.5.3 FUNCTION: TNPP_MAIN

As described, `tnpp_main()` is the central switchboard for skymds10. Coded in mdstnpp.c, `tnpp_main()` calls one of five functions held as a five member array of function pointers. Each function returns a status, which determines the next function called.

- an array of the five function pointers is declared
- the initial state is set to TNPP_INIT

The infinite message-processing loop starts here.

- The appropriate function based on the current status is called. These are all coded in mdstnpp.c.

Condition	Internal Constant	Value	Handler
(re) initialise the connection	TNPP_INIT	0	<code>tnpp_init_state()</code>
Wait for Remote communication	TNPP_AWAIT_ENQ	1	<code>tnpp_await_ENQ_state()</code>
Standard idle / poll state	TNPP_READY	2	<code>tnpp_ready_state()</code>
Transmit a message	TNPP_TX	3	<code>tnpp_transmit_state()</code>
Get a response from the remote site.	TNPP_AWAIT_TX_RESPONSE	4	<code>tnpp_await_tx_response_state()</code>

Table 191: skymds10: major behaviours and function calls [repeated]

- test the shut down semaphore. Time to die?
- go around the loop again, calling the appropriate function for the new status.

Note that, as each communication process communicates with the remote TNPP client that client does not directly set the new state. What it does is return one of many (i.e. >> 5) values. These are then converted to statuses by looking them up in the tstate array, a paired list of TNPP return values and statuses.

31.6 THE FIVE STATE-DEPENDENT TNPP PROCESSING FUNCTIONS

31.6.1 FUNCTION: TNPP_INIT_STATE

`tnpp_init_state()` returns skymds10 to the ‘initialise’ state.

- call `tnpp_clear_serial_table()`. The serial table is a simple array that keeps track of the messages pending and their order / sequence. This function cleans out the table, annihilating any messages pending.
- an ENQ character (ACSII 5) is sent to the remote TNPP system by `tnpp_send_host()`. This indicates an ‘enquiry’: are you there?
- the value TNPP_AWAIT_ENQ is returned, indicating to the calling `tnpp_main()` function that skymds10 is now awaiting the response to an ENQ / enquiry

31.6.2 FUNCTION: TNPP_AWAIT_ENQ_STATE

Well, as the name suggests, here we await the response from an ENQ / enquiry packet previously sent to the remote TNPP station.

- If we are on a simplex link, just return with the status TNPP_READY as we don’t want to wait on these type of connections
- set the maximum response time and call `tnpp_get_response()` to retrieve the response from our TNPP partner.
- if we are connected to a pagepoint system we keep trying. Then we give up and return a TNPP_INIT status, i.e. we have problems and should start again.

Now we send our next signal to the remote client (with `tnpp_send_host()`) and then read and interpret our previous response. `tnpp_send_host()` doesn’t seem to change the current status – as held in the tstate array – so it doesn’t matter which way around these statements appear. Possibly done this way in case any updating of status is done by the `tnpp_send_host()` invisibly or in the future. It’s unlikely though.

- the appropriate next message is determined from the tstate array and sent to the remote TNPP site by `tnpp_send_host()`
- then our (new) status is read from the goto_state property of the appropriate member of the tstate array. This tstate array is indexed by current state (5 of them) and most recent reply from the other TNPP site (more than 5)

31.6.3 FUNCTION: TNPP_READY_STATE

This is where we poll for incoming messages (from the external TNPP site) and send messages (to the TNPP site). From the function’s header comment:

“Idle state, need to poll for incoming messages as well as poll for outgoing messages. `tnpp_get_response()` performs a waited read for one second, if no message detected in this period it will look for any messages for dispatch. If a message is ready for dispatch it will return with IR_NRX early to allow this routine to terminate and subsequently the message will be dispatched.”

The last sentence is not quite correct. The IR_NRX code is converted to a TNPP_TX code by looking up the tstate array. It is the TNPP_TX code that is returned and that prompts the `tnpp_transmit_state()` function to be called. The sentence before that is sub-perfect as well: `dispatch_tnpp_packet()` is called to send a packet *first*, *then* we listen for incoming, for 10 seconds, not one second.

- call `dispatch_tnpp_packet()` obtains a messages from <System_Name> and sends it to a foreign host. First it uses `get_mess_from_ip()` (coded back in skymds10.c), which reads the message from the socket to obtain the message from the <System_Name> system. `dispatch_tnpp_packet()` then calls `write_tnpp()` to format and queue the message en route to the external TNPP system.
- `tnpp_get_response()` now listens for tnri_timeout (10) seconds for something from the TNPP site.
- use `tnpp_send_host()` to post a reply to our TNPP friend based on whatever was returned by the previous `tnpp_get_response()` call.
- Now check and return the status received from that call.

31.6.4 FUNCTION: TNPP TRANSMIT STATE

tnpp transmit state() calls either

- `tpp_transmit_null()` if the send_null flag has been set
 - `tpp_transmit_packet()` if not.

This flag is (only) set by the `tnpp_initialise()` system initialisation function, or after receiving and ACK (ASCII 6) from the external TNPP system.

- Transmission Failure:
 - TSTATUS_TRANSMIT_FAILED is passed to `tnpp_status()`. `tnpp_status()` acts appropriately on a range of responses from the external TNPP site.
 - `tnpp_transmit_state()` returns with the state TNPP_READY.
 - Thus skymds10 reverts to polling for the next message(s) to send / receive.
 - Transmission Success:
 - TSTATUS_TRANSMIT is passed to `tnpp_status()`. As just noted `tnpp_status()` acts appropriately on a range of responses from the external TNPP site.
 - `tnpp_transmit_state()` returns with the state TNPP_AWAIT_TX_RESPONSE.
 - We switch to waiting for the external site to respond to our transmission

31.6.5 FUNCTION: TNPP_AWAIT_TX_RESPONSE STATE

Here `skymds10` waits for a response from the external TNPP site. We will get here after transmitting a message (see `tnpp_transmit_state()`) and by repeatedly calling this function after running into problems.

- first we use `tnpp_get_response()` to extract a signal of some type from the remote TNPP site over the space of `tnri_timeout(10)` seconds. `tnpp_get_response()`:
 - first tests with `check_mess_from_ip()` whether there are any local messages to be sent. If there are it returns with the code `IR_NRX`, which tells skymds10 to call the `tnpp_await_ENQ_state()` function`.
 - If there are no messages found with `check_mess_from_ip()` `tnpp_get_response()` reads the response from the external TNPP site
 - The response we got, from either failure or success, is converted via the `tstate` array to the value that we actually want to send, and `tnpp_send_host()` is used to copy this value to the TNPP client.

The value we received is used as an index into the **tstate** array, an array of

```

(0) { value we actually want to use = 4, TNPP_* status / function constant C }
(1) { value we actually want to use = 1, TNPP_* status / function constant B }
(2) { value we actually want to use = 1, TNPP_* status / function constant C }
(3) { value we actually want to use = 3, TNPP_* status / function constant A }

        :
        :
        :
        :

(?) { value we actually want to use = 2, TNPP_* status / function constant A }

```

So if the value returned from `tnpp_get_response()` is (a constant that represents) the number 2, we actually use (i.e. forward to the remote TNPP site) the value '1' (lets say) and we set the `TNPP_*` status / function constant to 'C' (lets say).

Now there are 5 of these arrays, one for each of the current TNPP_* status / function constants, so there are five sets of collections of returned value [] (value we actually want to use + TNPP_* status / function constant) groups.

- now we set the TNPP_* status / function constant (e.g. TNPP_AWAIT_TX_RESPONSE) to some new value, as per the tstate array.
 - All done. Return the new status.

31.7 SUPPORT FUNCTIONS

31.7.1 FUNCTION: TNPP_SETUP

`tnpp_setup()` prepares and establishes a connection to the remote TNPP site.

- zero out the packet queue

The foreign client may be “supplying their own lun”. If so, then we don’t have to do these following steps setting up the port.

- First open the modem port in with no delay, to avoid hanging if the open fails. Once the dummy open is successful opened, it is reopened (and locked) properly in delay mode.

No-Delay mode forces the reading process (us, usually) to wait an arbitrary amount of time between packets. This is what we want to do; we don’t want to corrupt the communication every time there is some sort of delay. On the other hand, if the port can’t be opened for some reason we may wait forever. Hence the behaviour just described.

- set up the parameters and options for the port, all kinds of technical stuff.
- finally clean out the transmission buffer and return.

31.7.2 FUNCTION: TNPP_SEND_HOST

`tnpp_send_host()` is a simple wrapper for a `write()` call to the communications port we are using to connect to the external TNPP site. It takes the message character codes used by skymds10, converts them to the matching and appropriate ASCII codes, and writes this/these ASCII codes to the communications port.

The translations are as per the following table:

the label:	with the value:	is converted to:	which is:	with the value:	and means:
				(decimal ASCII)	
IR_NO_RESP	-1			---	No action or conversion taken, return success ---
IR_ENQ	0	ENQ_char	ENQ	5	Enquiry
IR_EOT	1	EOT_char	EOT	4	End of Transmission
IR_ACK	2	ACK_char	ACK	6	Acknowledge
IR_NAK	3	NAK_char	NAK	21	Negative Acknowledge
IR_CAN	4	CAN_char	CAN	24	Cancel. Can not process
IR_RS	5	RS_char	RSC	30	Sleep

Table 192: skymds10: Standard I/O port control characters and meanings

After the translation `tnpp_send_host()` simply writes the ASCII value to the port.

31.7.3 FUNCTION: TNPP_GET_RESPONSE

`tnpp_get_response()` reads a message from the external TNPP site. In a loop it looks for messages from either our (the <System_Name>) side or the TNPP client. If it finds a message from our side `tnpp_get_response()` exits so that another function can handle transmission of the message. Otherwise, if a message from the TNPP system appears the remainder of the function deals with reading in this message. It is built around two infinite loops, the inner one testing for incoming messages, the outer processing the message found.

skymds10 spend most of its time in this function, returning when not performing other tasks.

Inner Loop

If we are on a permanent connection (leased line), which we always are, we only stay in this inner loop for 10 seconds. However, if we are on a dial up we remain for ever, until we have some thing to send. Then we exit. No reading possible and no progress (i.e. hanging) until something to send. This is a bug. If this program is ever to be used on a dial up this will have to be changed.

Message from <System_Name>: `check_mess_from_ip()` reads the socket connection to the <System_Name> system. If a message is found on the socket, `tnpp_get_response()` exits with the code IR_NRX in order for other functions (`tnpp_transmit_state()` and `tnpp_transmit_packet()`) to deliver the data (from <System_Name> to the TNPP client).

- Message from external TNPP site: waiting for one second, any data on the communication port is read into a buffer
- The shut down semaphore is now tested. Time to die?
- We now decrement the `response_timeout` counter. `response_timeout` is initialised to 10 (seconds), as each pass takes 1 second, up to 14⁴⁴ attempts to read and send data will be made.

Outer Loop

We get here if there is something has been read over the communications port or we made it through 10 seconds without reading anything, in which case we also exit straight away. In this inner loop skymds10 interprets and handles whatever has been read from the external TNPP site.

- First, if nothing was read,
 - if time spent waiting pushes our complete idle time (`idle_timeout`) over 60 seconds (in mdstnpp.h the value of TIDLE is set to 60, `tidle_timeout = TIDLE`) send the value TSTATUS_IDLE to the `tstatus()` function, and return with status IR_IDLE
 - if we have not exceeded our timeout `tnpp_get_response()` calls `tstatus()` with the value TSTATUS_TIMEOUT and returns the status IR_NRX
- Now read the first character of the data received and act appropriately

Received:	<u>Process Messages</u>		<u>Send Null</u>		<u>Description</u>
	<u>tstatus() parameter</u>	<u>return value</u>	<u>tstatus() parameter</u>	<u>return value</u>	
ENQ(5)	TSTATUS_ENQ	IR_ENQ			respond to 'All-OK' <u>ENquiry</u>
EOT (4)	TSTATUS_EOT	IR_EOT			TNPP responding to our <u>ENquiry</u>
ACK (6)	TSTATUS_ACK	IR_ACK			<u>ACKnowledge</u> . No message
NAK (21)	TSTATUS_RETRY	IR_CRETRY	TSTATUS_NAK	IR_NAK	Negative ACK. Problem.
CAN (24)	TSTATUS_CAN	IR_CAN	---	IR_CAN	<u>CANcel</u> . Can not process.
RSC (30)	TSTATUS_HOLD	IR_CHOLD	TSTATUS_RS	IR_RS	Sleep for THOLD (10) seconds
SOH (1)	----	The start of a message. See the following description.	----	----	Message: Start of Header

Table 193: skymds10: Handling of standard I/O port control characters

Except SOH (Start of Header), a `tstatus()` call with parameter, then returning with the described value are pretty much all that is done.

The remainder of the function processes the data following a SOH character.

- Obtain the message: read the port until an ETX (End of Text – ASCII value 3) character is read.
- record any (other) STX (Start of Text – ASCII value 2), ETX, and / or SOH characters read.
- Read the CRC (Cyclic Redundancy Check) transmission integrity check value
- if the message is too long (buffer overflow) then truncate the message.

Two things. The bad news: if this is protection against a buffer overflow attack then it's too late. The good news: but its not too late – this code will never be executed. The read loop reading the CRC value is limited to `TNPP_BUFFER_SIZE` many single character reads.

- Now recalculate the CRC and check that it matches the value sent with the message. Also do a couple of other simple idiot-checks
- Read the serial number and add it to the serial number table. This table tracks all our messages-in-progress.
- Now check the address. It must be properly addressed (to <System_Name>). If it is not we return IR_UGLY which will send a CAN (cancel. Can not process.) character back to the non-local <System_Name> site.
- Finally check in case we have received an empty (null) message when we weren't expecting one. If so we handle it

If we made it to here we have received a good message.

⁴⁴

- copy the message into the `recv_queue_buffer()`
- finally we pass TSTATUS RECEIVED to `tstatus()`. While `tstatus()` usually only needs to update the return code, if one passes it TSTATUS RECEIVED `process_incomming_packet()` (coded in mdstnsub.c) is called which copies the message to the bulk queue.

31.7.4 FUNCTION: DISPATCH_TNPP_PACKET

`dispatch_tnpp_packet()` is the interface for obtaining a message from <System_Name> and sending it to the remote TNPP system. `dispatch_tnpp_packet()` is a very simple function itself, implementing its functionality via only two function calls. This description is more a look-under-the-hood.

`dispatch_tnpp_packet()` calls

- `get_mess_from_ip()`, which obtains a <System_Name> message via a socket to a skymds03 program
- `write_tnpp()`

then:

- `write_tnpp()` calls `tnpp_queue_packet()`
- `tnpp_queue_packet()` copies the message to the wait_queue_buffer.
- (shortly later) `tnpp_transmit_packet()` copies the message out of the wait_queue_buffer into the send_queue_buffer from where it is written in MAX_OUTPUT_SIZE chunks to the TNPP communications port.

31.7.5 FUNCTION: CHECK_MESS_FROM_IP

`check_mess_from_ip()` tests the socket that connects skymds10 to skymds03, the <System_Name> message delivery daemon, setting the global `read_socket_before` flag to true if there is data to be read from the socket.

- call the Unix system function `recvfrom()` to read any data from the socket into a buffer and return the length of the stream of data written
- deal with socket errors and exit if no data is waiting.
- use the Unix system function `gethostbyaddress()` to identify where the data is coming from (what IP address)
- `read_socket_before` is set to true, indicating that there is data waiting to be read

31.7.6 FUNCTION: GET_MESS_FROM_IP

`get_mess_from_ip()` checks for and obtains a message from <System_Name> over a socket established with a skymds03 message distribution daemon. In checking for messages the code from the `check_mess_from_ip()` is pasted in virtually in its entirety.

- if this instance of skymds10 is not in “listen mode”, that is, the -l command-line parameter was not used, we exit out straightaway.
- now, as noted, the code of the `check_mess_from_ip()` function is pasted in effectively complete. A simple function call would probably have done.

The message has now been read into the message buffer, either by a previous `check_mess_from_ip()` call or by the duplicate of it just passed through.

- the `read_socket_before` flag is set to false.
- the data from the message read is copied into a structure to be returned to the calling process
- call `is_message_duplicated()`
- finally `read_pager()` is used to check that the pager number indicated on the message actually exists in the mdspager database table.

There is a note in the source code that this last function call (`read_pager()`) should be removed.

31.7.7 FUNCTION: TNPP_TRANSMIT_PACKET

`tnpp_transmit_packet()` reads data from either the send or the wait queue to the communications port <Company_Name>ing us with the external TNPP site. Any data in the send queue takes priority over the wait queue, though data left over here indicates some sort of problem previously.

If nothing in the send queue (as it should be):

- if nothing in the wait queue (either) then nothing to do: exit.
- so there *is* something in the wait queue: copy it to the send queue.
- set the length of the send queue to what the wait queue used to be, and set the length of the wait queue to zero.
- if the last character is the ASCII value 23 (octal 27), i.e. End-Transmission-Buffer (ETB), then chop it off.
- prepare the data in the send queue for posting:
 - calculate the message length,
 - CRC⁴⁵ value,
 - add a trailing null character,
 - save a copy of it. This is used in case we have to re-send (see below)
- write the data to the communications port

done. Now the (hopefully rare) case of send-buffer data left over from the previous time around.

- copy the data out of the copy we saved previously
- goto the last step above, the writing of the data to the communications port.

really done.

31.7.8 FUNCTION: TNPP_TRANSMIT_NULL

`tnpp_transmit_null()` is a stripped-down version of `tnpp_transmit_packet()` that sends null messages, complete with serial-number and CRC values, to the foreign TNPP site. It doesn't read or interact with the send or wait queues and, excepting a port-error, should always succeed in its simple task.

31.7.9 FUNCTION: TNPP_STATUS

`tnpp_status()` actions any immediate, standard responses to be taken each time the external TNPP site changes its 'state'. Note that these are only a part, possibly a small part, of the response / message handling by skymds10 of TNPP data. These are the standardised, uniform, automatic responses. Other parts of skymds10 supply other, further processing of particular status / message conditions.

The most important part of this function is the TSTATUS_RECEIVED handling routine, which implements the receiving half of skymds10's functionality.

`reply_to_host()` sends a message back to the skymds03 program by which (this) skymds10 daemon is connected to the <System_Name> system as a whole.

31.7.9.1 On receipt of a packet from the external TNPP site

On receipt of a TSTATUS_RECEIVED constant (value = 2) `tnpp_status()`:

- calls `tnpp_dequeue_packet()` to copy the message from the 'in buffer' to a message structure
- `process_incomming_packet()`, coded in mdstnsup.c, writes the packet to the bulk queue.

This is obviously an important part of skymds10 as a whole.

⁴⁵ CRC: 'cyclic redundancy check' value, a check sum used to detect en-route corruptions.

31.7.9.2 Other responses from the external TNPP site

These are the other responses `tnpp_status()` makes to status changes in the TNPP client. Basically they all consist of calling `reply_to_host()` to pass a message back to skymds03, and optionally flagging the message as sent.

Status / condition	Constant	Action
Exceeded Max No of retries (to send) TNPP site says: "Discard this packet."	7: TSTATUS_RETRY 5: TSTATUS_CAN	<ul style="list-style-type: none"> • <code>reply_to_host()</code> • flag as sent, i.e. don't (re)-send.
TNPP site has received a packet	3: TSTATUS_ACK 8: TSTATUS_HOLD	<ul style="list-style-type: none"> • <code>reply_to_host()</code>
Lost connection	1: TSTATUS_DOWN	<ul style="list-style-type: none"> • test whether to raise an alarm • set the tnpp_abort flag to true
TNPP site says: "Take a short break"	6: TSTATUS_RS	

Table 194: skymds10: non-message TNPP responses: processing

31.7.9.3 'Don't respond' status conditions

Status / condition	Constant	Action
Newly connected	0: TSTATUS_UP	
We have received a NULL packet	13: TSTATUS_RX_NULL	Nothing. Log a message under debug
Have reached the idle-time limit.	10: TSTATUS_IDLE	
End-of-transmission	14: TSTATUS_EOT	
Enquiry ("Are you there?") received	15: TSTATUS_ENQ	Nothing
Have timed-out somewhere.	9: TSTATUS_TIMEOUT	
We have received an Error indicator.	4: TSTATUS_NAK	

Table 195: skymds10: non-message TNPP responses: ignore

31.7.10 FUNCTION: PROCESS_INCOMMING_PACKET

This function, coded in mdstnsub.c, writes a message from a TNPP message structure to the <System_Name> bulk queue. It deals with three types of TNPP messages: TNPP format, Cap code format, and TNPP command

TNPP pager message format

- get the pager number (which may be other things than an actual *pager* number) and use `read_pager()` to check that it is OK.
- copy the message into a buffer and append a NULL character to the end of it
- now the function digit is copied from either the TNPP message or, if not present, from the appropriate pager record.
- call `setup_queue_rec()` to prepare a bulk queue record.

Cap Code Format

- get the pager number (which may be other things than an actual *pager* number) and use `read_pager()` to check that it is OK.
- copy the message into a buffer and append a NULL character to the end of it.
- do a conversion of message formats if necessary. Use `cvt_numeric_mess()` to do the conversion.
- get the signalling and function digit values from the message
- call `setup_queue_rec()` to prepare a bulk queue record.

TNPP command Format

This makes a few calls, but at the bottom of it, well, it's just a placeholder. **A()** calls **B()** which calls **C()** which has a comments along the lines of: "If more clients (than the Sth Australian Govt) start to use this program, then put additional code in here

Now ...

- Truncate the message if it is too long
- call **queue_main()** to post the message to the bulk queue
- have a brief sleep to stop messages piling on top of each other

Done.

31.8 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

31.9 STOPPING SKYMD10

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)

32 skymds11t (the former skymds11a)

32.1 SKYMDS11T OVERVIEW

skymds11t transmits SMS messages from <System_Name> to an external SMS Centre (SMSC). Its partner in SMS, skymds11r, receives the replies back from the SMSC for filing in the <System_Name> database and billing systems.

skymds11t was formerly known as skymds11a, while skymds11r is the former skymds67a. At the time of writing skymds11a has not yet been changed to skymds11t though skymds67a has successfully reappeared as skymds11r.

32.2 RUNNING SKYMDST11T

skymds11t runs on the servers lnka (in Adelaide), lnko (in Melbourne), and lnks (in Sydney). lnkx (the secondary Sydney server) may be added. At each of these sites skymds11t runs as a daemon, being executed, like most <System_Name> programs, by the start.mds script. At two of these sites the person running the start.mds start-up script is prompted: “Run these programs … (y/n)” type thing, but the list of programs displayed still contains the no longer existent skymds67a. Different prompts may be constructed soon.

32.2.1 COMMAND LINE PARAMETERS

Parameter	Description
<u>Mandatory</u>	
-t < port >	communications port for skymds11t to be recorded against in the mdspports table.
-l < port >	the port on which to listen for messages from skymds03
-d < SMS dest >	Specified SMS destination
<u>Optional</u>	
-D	run in Debug mode, listing detailed run-time information.
-a <address range >	address range
-p	Give SMS message priority
-c	use country codes.
-r	send SMS by ‘registered’ delivery – return receipts
-B	‘Bureau mode’: no two-way messaging
-C < code >	Carrier code
-?	print usage instructions and exit.

Table 196: skymds11t: Command Line parameters

32.2.2 EXAMPLE COMMAND-LINE

On the Sydney lnks server three copies of skymds11t are run, two in debug mode. Note that it is still the previously named skymds11a used in these examples, soon to be replaced by skymds11t.

```
$MDS_BIN_DIR/skymds11a -D -t /dev/dum59 -d VODA_MOBMG -l 3456 -c 2>>$MDS_LOG_DIR/mds11avod.log  
$MDS_BIN_DIR/skymds11a      -t /dev/dum54 -d SMSC_VODA  -l 3455 -r 2>>$LOG_SCR
```

The first is run:

- in debug mode
- registered in mdspports against the port /dev/dum59
- send to the Vodafone mobile message network (in the table mdsnetcn, VODA_MOBMG = “voda_mobimesg”)
- listening for SMS messages from skymds03 on the port 3456
- using country codes
- with stderr (file descriptor 2) copied to the file MDS_LOG_DIR/mds11avod.log

The second is run:

- registered in mdspports against the port /dev/dum54
- sending to the Vodafone SMSC (SMSC_VODA)
- listening for SMS messages from skymds03 on the port 3455
- sending by registered SMS mail – return receipts requested.
- with stderr (file descriptor 2) copied to LOG_SCR

32.3 PROGRAM DESCRIPTION

skymds11t is the <System_Name> SMS transmission program. After initialisation control passes to `process_smpp()` which runs as a double infinite loop. The outer loop (re) establishes the network connection each time there is a problem, while the inner loop gets the next message from its host, sending to the foreign SMSC, and sending an “OK – done” back to its host.

32.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds11t.

32.4.1 SOURCES FILES

Source File	Description
skymds11t.c	SMPP SERVER for SMPP V3.4. (modified for SAU)
gensubs.c	Contains various general-purpose routines that are separately compiled.
isstart.c	Wrapper for C-ISAM isstart function – choose index and position the read-point
mdscdr.c	Contains various general-purpose routines that are separately compiled.
mdscomio.c	Various serial io routines
mdscomm.c	Socket io and Serial module.
mdsginit.c	A number of functions that perform commonly used initialisation code segments.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the ports file.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdssemcck.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssmppa.c	Communications functions related to communications to a SMSC using the SMPP protocol.
mdstmrtn.c	Contains functions, which perform operations to do with time.

Table 197: skymds11t: Source Files

32.4.2 HEADER FILES

Header File	Description
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.: C time field definitions
macros.h	User defined macros.: definitions & macros for common functions
mds2wsb.h	2 way bulk-email send file
mds2wse.h	2 way email send file
mdscdr.h	CDR definitions file
mdscdrec.h	CDR data structure
mdsfldsz.h	Standard <System_Name> includes.: field length & definitions
mdsncrec.h	Network Connection master file
mdsparam.h	parameters used by PMS system
mdsporec.h	Ports security master file: Ports Record
mdsquec.h	Contacts Message queue file: Destination Message queue file
mdsrerec.h	Record reply back to remote host
mdssmppa.h	SMPP definitions.

Table 198: skymds11t: Header Files

32.4.3 DATABASE TABLES

Each of these header files contains (only) a C-struct that matched the structure of a database table used by skymds11t.

Database Table	Header File	Description
mdscdrec	mdscdrec.h	Call details master
mdsnetcn	mdsncrec.h	Network connection
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsquec.h	Destination queues

Table 199: skymds11t: Database tables

32.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description	Example
MDS_DATA_DIR	data directories, queues, database and history.	/var/mds/data
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide ...	O
PASSWORD	password to access the SMSC	8ashgxca87c6acigak...
SERVICE_TYPE		‘‘
SOURCE_ADDR		921050
SOURCE_ADDR_NPI		3
SOURCE_ADDR_TON		2
SYSTEM_ID		<Company_Name>
SYSTEM_TYPE		<Company_Name>comm

Table 200: skymds11t: Environment Variables

32.5 SKYMDS11T: MAJOR FUNCTIONS

32.5.1 FUNCTION: MAIN

- **initialise()** prepares skymds11t's run-time environment
- **process_smpp()** performs the work of skymds11t, reading SMS messages from the <System_Name> system and passing them to one or more non-<Company_Name> SMSCs
- upon program termination finalise is used to close all open database tables.

32.5.2 FUNCTION: INITIALISE

The **initialise()** function sets up skymds11t's run-time environment, via the following steps:

- reads the environment variables needed for the program
- parse the command line
- **initialise_tlv_options()**, coded in mdssmppa.c, is now used to allocate space for the TLV variables. TLV variables are used in the transmission of SMS messages.
- **background()** is then called to run skymds11t in the background as a daemon.
- catch a few signals
- use **PORT_set_port_started()** to log skymds11t against the port passed to skymds11t on the command-line. This is done for automatic monitoring of the <System_Name> system by mdsalarm.
- access the shutdown semaphore
- open the necessary database tables.

The environment variables as listed above are read in from the environment.

32.5.3 FUNCTION: PROCESS_SMPP

This is the work function of skymds11t, reading the messages from an instantiation of skymds03 and passing them to an SMSC. It runs in a double infinite loop, the outer loop maintaining and re-establishing connections to the SMSC, the inner loop sending messages to the appropriate SMSC

The outer loop – connecting to an SMSC.

- use `PORT_update_port_rec()` to note any new connection details
- `connect_and_bind_to_smssc()` (coded in mdssmppa.c) attempts to connect to the SMSC.

`connect_and_bind_to_smssc()` calls `init_tcpip_connection()` or `init_x25_connection()` to open a socket connection to the SMSC. If `connect_and_bind_to_smssc()` fails, we have to switch to another telecommunications carrier's SMSC. The next few commands deal with this.

- if `connect_and_bind_to_smssc()` has failed `GEN_get_next_netw_connect()` is tried and we start the outer loop again
- if that fails `GEN_get_netw_connect()` is used. If that fails its all over and we break-out
- But if all is swell we now do the other side: call `connect_to_lan()` to establish our connection to skymds03. skymds03 is the source of all our SMS messages.

The inner loop – obtaining and posting SMS messages .

- first check the shutdown semaphore
- Now is an all-OK check. If we have to restart the loop four times without doing anything `send_enquire_<Company_Name>()` is used to ensure that all is just fine.
- `wait_either_input()` waits for up to 10 seconds for an indication of a SMS message, either from the SMSC or from the <System_Name> network. If this fails `check_enquire_<Company_Name>()` tests the connection. If `check_enquire_<Company_Name>()` also fails we break to the outer loop to reinitiate the <Company_Name>, else we just restart the inner loop, waiting for the next SMS message.
- If we have made it this far we can now obtain the SMS message (from wherever) by `get_next_mess()`.

Having got the message skymds11t now prepares to port it to its destination.

- `check_duplicate_mess()`, as its name suggests, ensures that we haven't seen this same message before
- `get_smpp_source_address()` and `setup_submit_packet()` prepare the SMS for delivery
- `mess_to_smssc()` delivers the message

And now do some clean-up:

- record some message details in the host_list array
- `reply_to_host()` writes an OK message back to the SMS message's originator.
- if we are not using our first choice SMSC, do the switch now, if possible.

Back to the start of the inner loop.

End of outer loop:

Back to maintaining the network connections. If we have got to here we must have broken out of the inner loop with either some sort of network problem or program termination, so we close the network connections and re-open them at the start of the next loop... or maybe never.

- `close_connection_smssc()` well, closes the connection to the SMSC
- `close_lan()`, well, closes the connection to the LAN.
- again check the shutdown semaphore. If it has been set to 0, then it is time to die.

32.5.4 FUNCTION: CONNECT_AND_BIND_TO_SMSC

`connect_and_bind_to_smsc()` sets up a TCP/IP connection to the SMSC.

- first the tcp/ip binding structure is prepared.
- then in an infinite loop either setup an X.25 or a TCP/IP connection depending of the flags set
- now send a test message to the SMSC via **`mess_to_smsc()`** and test the response.
- If the repines was OK, then all done → exit. If it was not OK, the connection is closed and we go around again.

This function continues looping until a connection is made.

32.5.5 FUNCTION: CONNECT_TO_LAN

`connect_to_lan()` sets up the skymds11t end of the socket on which it listens for messages from skymds03.

- populate a sockaddr_in structure with details of the skymds11t end of the socket
- call **`socket()`** to create the socket
- then **`bind()`** configures the socket with the parameters entered into the sockaddr_in structure in the first step above.

32.5.6 FUNCTION: WAIT_EITHER_INPUT

This function tests the two message sources (the <System_Name> LAN and the foreign SMSC) for activity, using the **`select()`** function and the FD_SET and FD_ISSET macros to build a list of data sources: the socket to skymds03 and the special block file (port) connection to the SMSC. These macros are designed to be used with **`select()`** for this purpose.

- using macros (FD_SET) the file IDs for the socket connection to skymds03 and the port connection to the smpp are added to a **`fd_set`** structure.
- The **`fd_set`** structure is passed to **`select()`**, which returns how many (of the two) files have messages in them.
- If the number is greater than zero, the FD_ISSET macro is used to determine which file is active. Depending on this result the appropriate flag is returned to the calling process

Note that this function will handle arbitrary and changing external connection systems.

32.5.7 FUNCTION: CHECK_ENQUIRE_<COMPANY_NAME>

`check_enquire_<Company_Name>()` is a short function to test the status of the connection to skymds11t's SMSC.

- **`smpp_recv()`** reads a packet from the SMCC. timeouts are treated as success
- other than timeouts, anything other than SMPP_E_OK (message OK, value 0x00) is treated as an error.
- we now test the value of **`smpp_enquire_<Company_Name>_resp()`** when it sends a message to the SMSC.
- again, anything other than SMPP_E_OK is an error.

32.5.8 FUNCTION: GET_NEXT_MESS

`get_next_mess()` is basically a wrapper for the Unix **`recvfrom()`** command, reading a message from a socket

- **`recvfrom()`** extracts REMOTE_SMS_2WAY_SIZE bytes of data from the socket
- most of the remainder of the function is spent coping parts of this stream of data to a message structure
- the host name is determined by using the **`gethostbyaddr()`** function, which supplies the name from the /etc/hosts file.

32.5.9 FUNCTION: MESS_TO_SMSC

`mess_to_smsc()`, coded in mdssmppa.c, sends the message to the SMSC. It actually uses `smp_send()` (also in mdssmppa.c) to actually deliver the package.

- first `smp_send()` is used to deliver the packet
- then `smp_recv()` is used to test the response from the SMSC
- the result is tested as to whether it is SMPP_E_OK and that the message has the correct sequence number.

32.5.10 FUNCTION: REPLY_TO_HOST

`reply_to_host` sends a message (back across a socket connection) to skymds03. It is basically a wrapper for the Unix function `sendto()`. skymds11t's partner program, skymds11r, is the main daemon for sending messages from the SMSC end to the skymds03 end of this system.

- first the message string is filled up with appropriate data
- `sendto()` is then used to send it across the socket to skymds03.
- the remainder of the function is concerned with a small amount of error handling and debug-mode dumping of the data.

32.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

32.7 STOPPING SKYMDS11T

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)

33 skymds27

33.1 SKYMDS27 OVERVIEW

“Contacts” are the <System_Name> messages requiring advanced handling. These messages (“contacts”) pass through the bulk queue, to skymds20, which passes them on to this program – skymds27 – to implement the advanced handling. Currently the work of skymds27 consists of:

Contact	Description
Advisories	Messages to callers, with their (simple) collected responses
Call Patches	Connecting a caller to a destination. Like a telephone operator.
Escalations	Change the method of message delivery until receipt is acknowledged
Emails	Send an email from <System_Name> to an external destination
Faxes	Send a fax
Forms	Process the data entered on a customised form
Group Paging	Send a pager message to multiple pagers
Paging – Standard paging	Send a normal pager message
TMS – answer phone	Record a message from the user
Text processing (email)	
Reminders	Add a reminder message to the Call Centre’s reminder queue
Reminders: Supervisor Reminder	Add a reminder for a Call Centre supervisor to read and act on
Reminders: Manual Reminders	No processing required for manual reminders
Reminders: Phone Dummy Rem’rs	Send a reminder into the void – for some reason
Rosters	Outside certain times apply alternative processing to the message.

Table 201: skymds27: ‘Contact’ types and Descriptions

Messages representing contacts of each of these types are delivered to skymds27, which, calling one of a number of sub-functions, processes the contacts appropriately. ‘processing the contacts appropriately’ can involves:

- sending one or more pager or contact messages back to the bulk queue: if it is a contact it ends up, in its new form, in skymds27 again.
- generating new simple pager or contact messages and iteratively processing these within skymds27.
- posting the processed message to its final destination, e.g. the fax queue
- posting the processed message to the maintenance queue.

33.2 RUNNING SKYMDS27

skymds27 is run from the <System_Name> start.mds start-up script.

33.2.1 COMMAND LINE PARAMETERS

Parameter	Description
-t < port >	the communications port skymds27 is registered against
-l < pc->	
<Company_Name>	
-d < queue >	the short name of destination message sub queue for skymds27 to read.
-D	run in debug mode, listing detailed run-time information.
-?	print usage instructions

Table 202: skymds27: Command Line Parameters

33.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute skymds27 at one <System_Name> site is:

```
$MDS_BIN_DIR/skymds27 -t /dev/dum36 -d CONT -l 96252082 2>$LOG_SCR
```

indicating that skymds27 is run:

- registered against port /dev/dum36 in the mdsparts database table
- reading messages from the CONT contacts message queue – as expected

log messages to stderr are redirected to `LOG_SCR`, #define'd in start.env to be `/dev/tty0p5`, a monitor somewhere.

33.3 PROGRAM DESCRIPTION

skymds27 is a daemon run to implement all the contact (advanced message) handling, the top-level logic being in its `main()` function loop. Here, after reading each message from the contacts queue, processes the message and calls `process_contact()` to manage the implementation of the processed message. `process_contact()` is basically a long switch statement that passes control to one of 14 `type_logic()` or `process_type()` functions, depending on the type of contact. Two types of contact don't require any processing. `process_type()` functions are wrappers for matching `type_logic()` functions.

Each of the `type_logic()` functions are coded in a separate mds27????.c file and <Company_Name>ed to skymds27 at compile time. The details of the current contact are stored in the (external, generally) structure "global_data" viz.:

```
global_data
{
    CONTACT_ID    parent_contact_id;
    CONTACT_ID    contact_id;
    CONTACT_TYPE   contact_type;
    EVENT_NUM     event_num;
    SYS_REF_NUM   sys_ref_num;
    int           mess_len;
    MESSAGE        message;
    MESSAGE        expanded_msg;
    char          hist_tran_type;
    CONTACT_ID    hist_contact_id;
    int           trans_complete;
    int           log_hist;
    int           err_type;
    FORM3         std_code;
};
```

Watch out though. There is another 'global_data' structure declared in mdssende.c, for "Email Initiator" usages.

33.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds27.

33.4.1 SOURCES FILES

Source File	Description
skymds27.c	Processes contacts (Rosters Escalations TMS etc) from the contacts queue file.
alarmrtn.c	Generic alarm routine.
crcalc.c	Calculate the CRC for a given string.
gensubs.c	Contains various general-purpose routines that are separately compiled.
globvar.c	Functions to access the Global Variable shared memory used by multiple programs.
isstart.c	Wrapper for C-ISAM isstart function – choose index and position the read-point
mds27adv.c	Advisory Clearing Function.
mds27cpa.c	Call Patch Clearing Function.
mds27eml.c	Email Clearing Function.

Table 203: skymds27: Source Files

Source File	Description
mds27esc.c	Escalation Clearing Function.
mds27fax.c	FAX Clearing Function.
mds27frm.c	Forms Clearing Function.
mds27grp.c	Groups Clearing Function.
mds27pag.c	Pager Clearing Function.
mds27phn.c	Phone Clearing Function.
mds27rem.c	Reminders Clearing Function.
mds27ros.c	Roster Clearing Function.
mds27tms.c	TMS Clearing Function.
mds27txt.c	Text Clearing Function.
mdscdr.c	Contains various general-purpose routines that are separately compiled.
mdsestats.c	Increment 'X-has-happened' count in a program statistics log file
mdsemqmain.c	Generic message email queuing routine.
mdsginit.c	A number of functions that perform commonly used initialisation code segments.
mdshisio.c	Generic history file handling routines.
mdskbld.c	Contains functions to build and unbuild pager keys, and set-up a dummy pager.
mdslog.c	Generic error logging.
mdsphol.c	Functions to attach, check and modify Public Holiday shared memory. Used by multiple programs.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurtn.c	Generic message queuing routine.
mdsrem.c	Functions to access the reminders queue, used by multiple programs
mdssetup.c	Obtain set-up definitions from a text set-up file.
mdstamp.c	Date time stamp & sequence number generation
mdstmrtn.c	Contains functions, which perform operations to do with time.
mdstxrtn.c	A number of functions, which perform commonly, used initialisation code segments.
mdsuniqn.c	Functions to get and update the Unique Number files

Table 204: skymds27: Source Files (continued)

33.4.2 HEADER FILES

Header File	Description
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.
globvar.h	Global shared memory structures
macros.h	User defined macros.: definitions & macros for common functions
mds27par.h	A global variable structure for contact processing,
mdsaprec.h	A-Party Master file
mdscdr.h	CDR definitions file
mdscdrec.h	CDR data structure
mdscorec.h	(CLT/INV/GRP) Contacts Master File
mdscprec.h	Call Patching Master File
mdcsrec.h	Client stats master file
mdctrec.h	Extended Canned Text File
mdsderec.h	Ports security master file
mdsecrec.h	External Contact master file
mdsemrec.h	Email File (Online & Batch)
mdsesrec.h	(ESC) Escalations Master File
mdsetrec.h	Escalation transaction master file
mdsflsz.h	Standard <System_Name> includes.: field length & definitions
mdsfmrec.h	(FRM) Forms Master File
mdsfxrec.h	(FAX) Fax Master File
mdshirec.h	History primary recs file
mdshsrec.h	Secondary History file (audits)

Table 205: skymds27: Header Files

Header File	Description
mdslirec.h	History primary file
mdsnrrec.h	New Rosters Master File
mdsparam.h	parameters used by PMS system
mdsparec.h	Pager Master File
mdsphol.h	Public Holiday Shared Memory file
mdsphtrec.h	(TMS) Telephone Message Service
mdsporec.h	Ports security master file: Ports Record
mdspstrec.h	Password master file
mdsqrec.h	Contacts Message queue file: Destination Message queue file
mdsrem.h	Reminders shared memory definition
mdsrmrec.h	Reminders master file
mdsterec.h	Timed event file (incl. booked call)
mdstxrec.h	(TXT) Delivery Stats Master File
mdsuniqn.h	Unique No. function parameter structure
mdsunrec.h	Unique Number File

Table 206: skymds27: Header Files (continued)

33.4.3 DATABASE TABLES

Each of these header files contains (only) a C-struct that matched the structure of a database table used by skymds27.

Database Table	Header File	Description
hiccyymmdd	mdshirec.h	the 'logical' history
hsyymmdd	mdshsrec.h	History secondary storage
mdsaparty	mdsaprec.h	A-Party Customer Info
mdscallp	mdscprec.h	Call patching master
mdscdrec	mdscdrec.h	Call details master
mdsclsts	mdscsrec.h	Client stats master
mdsconta	mdscorec.h	Contacts master
mdscetxt	mdstrec.h	Escalation large canned text
mdsdestm	mdsderec.h	Destinations master
mdsemail	mdsemrec.h	e-mail master
mdsescal	mdsesrec.h	Escalation master
mdsestrn	mdsetrec.h	Escalation transaction master
mdsextco	mdsecprec.h	External contact master
mdsfax	mdsfxrec.h	Fax master
mdsforms	mdsfmrec.h	Forms
mdsline	mdslirec.h	Indial number information
mdsnrost	mdsnrrec.h	Roster master
mdspager	mdsparec.h	Pager master
mdspassw	mdspstrec.h	Password username
mdsphone	mdsphtrec.h	Phone master
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsqrec.h	Destination queues
mdsrem	mdsrmrec.h	Reminders queue
mdstext	mdstxrec.h	Text header
mdstimev	mdsterec.h	Timed events
mdsuniqn	mdsunrec.h	Unique number

Table 207: skymds27: Database Tables

33.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description	Example
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide ...	O
MDS_DATA_DIR	The root of the data directories, queues, database and history.	/var/mds/data
MDS_LOG_DIR	the directory for logs to be written to	/var/mds/data/logs
MDS_BIN_DIR	All the <System_Name> executables, incl start-up scripts	/opt/mds/bin
MDS_TEXT_DIR	Message storage directory	/var/mds/data/text

Table 208: skymds27: Environment Variables

33.5 SKYMD27: MAJOR FUNCTIONS

33.5.1 FUNCTION: MAIN

As described above, the `main()` function collects each contact message from the CONT queue, pre-processes that message and passes it to `process_contact()` for handling and appropriate redirection.

- call `initialise()` to set up the skymds27's run-time environment.
- call `CDR_start_session()` to ensure that the CDR file is open
- now start the get message, process it, send it to `process_contact()` loop
- test the shut-down semaphore. The time to shut down is at the beginning of each loop.
- get the next message with `get_next_mess()`. If there are no messages then sleep for `sleep_time` seconds, sleep-time being stipulated on the destination queue database record for the queue that skymds27 is reading. The current 'delay' (i.e. `sleep_time`) setting for the CONT queue is 120, that is, 2 minutes. If there are no contact messages then sleep for two minutes.
- copy the destination queue message just obtained to a bulk queue structure and do some simple work on it.
- call `process_contact()` to send the contact message to its appropriate handler
- update the distribution queue header and go around the loop again

Once the loop is broken, `CDR_end_session()` closes the file pointer to the CDR file while `finalise()` closes all the database tables.

33.5.2 FUNCTION: INITIALISE

The `initialise()` function sets up skymds27's run-time environment, via the following steps:

- Read in the appropriate environment variables. See the table above
- parse the command line. Also see above.
- call `background()` to put skymds27 into the background.
- catch / redirect a few signals
- open the destination queue database table (mdsdestm), get the queue we should be reading (probably "CONT"), and open it

The `sleep_time`, primary index are obtained, and compare the port passed in on the command-line to that stipulated for the queue in the database. Finally flag in the mdsdestm that skymds27 has started.

- obtain the shutdown semaphore (from the mdssports table) and test it
- open all (21) of the database tables we need to use
- open up file pointers to the bulk, destination and maintenance queues
- shared memory: connect to the reminder, global, and public holiday shared memory segments
- set up flags in the maintenance flags shared memory structure.

33.5.3 FUNCTION: PROCESS_CONTACT

Process contract is just a switchboard, passing control to the appropriate contact handler. It tests the contact_type field of the “global” structure and calls the appropriate function as per the table.

The `process_type()` functions are wrappers for the matching `type_logic()` function. Each of the `type_logic()` functions are coded in their own mds27code.c file, e.g. mds27pag.c for `pager_logic()`.

As many types of contacts can be changed to other contact types as part or all of their processing, the passing of the contact message to these functions is looped until the trans_complete flag in the global shared memory is set.

contact_type Code	Function Called	Contact Type
ROS	<code>process_ros()</code>	Rosters – alternative processing based on time and date
ESC	<code>process_esc()</code>	Escalations – change processing until acknowledgement
PAG	<code>pager_logic()</code>	Paging
GRP	<code>process_grp()</code>	Group Paging
FAX	<code>fax_logic()</code>	Fax
FRM	<code>process_frm()</code>	Forms
TMS	<code>tms_logic()</code>	Telephone Message Service
TXT	<code>txt_logic()</code>	
REM	<code>reminder_logic()</code>	Reminders
PHN	<code>phone_logic()</code>	
EML	<code>email_logic()</code>	Email
SPV	<code>supervisor_reminder()</code>	Reminders to Call Centre supervisors
ADV	<code>adv_logic()</code>	Advisories
CPA	<code>cpa_logic()</code>	Call patch
MAN	No function called	Manual Reminders
DUM	No function called	Dummy Contact

Table 209: skymds27: Contact types, codes, and handlers

33.5.4 FUNCTIONS: PROCESS_ROS AND ROSTER_LOGIC

Rosters stipulate alternative handling of messages during set periods of time, i.e. they are rosters, or timetables.

`process_ros()` is a simple wrapper for `roster_logic()`, adding no further functionality and, in reality, is superfluous.

33.5.4.1 Function: roster_logic

- first `get_rosters()`, which calls `save_in_memory()`, is used to read all the rosters into memory. If there are no rosters available (and a roster contact message has been sent) then change this contact type to “SPV” (supervisor reminder) and return, as there is obviously some mistake.
- `find_active_roster()` loops through the list of rosters until an active one is found, that is, a roster set up such that we are currently in a period where the roster stipulates alternative processing. For each roster record `find_active_roster()` calls either `is_inc_roster_active()` or `is_excl_roster_active()` to test for the roster being active as either an inclusive or exclusive roster.

Rosters can be of two types, “inclusive” and “exclusive”.

- Inclusive: alternative handling from the start-time on the start day to the end time on the end day.
- Exclusive: alternative handling between start and end time on each day between start day and end day
- if `find_active_roster()` has found a currently applicable (‘active’) roster, the contact type of the contact message passed into `roster_logic()` is changed to the alternative processing type as stipulated by the roster record
- the contact is then returned to `process_contact()` to be delivered via its new contact delivery mechanism

33.5.5 FUNCTIONS: PROCESS_ESC AND ESCALATION_LOGIC

Escalations are messages that change their delivery mode until a successful acknowledgement of message receipt is received. **process_esc()** is coded in skymds27 while **escalation_logic()**, with its supporting functions, is in its own mds27esc.c file.

process_esc() :

- Assigns a unique ‘event number’ to the escalation about to be handled. (In <System_Name> lingo an ‘event’ is a handled escalation.)
- calls **escalation_logic()** to handle the escalation

33.5.5.1 Function: escalation_logic

- check that everything is OK with the escalation’s set-up. Check the escalation header record (in mdsestrn) is present and that the assigned event-number hasn’t been acknowledged yet.

A pager from a group:

- if the (escalation) contact message originally came from a group then this escalation message will need a new event-number for itself.
- a history record is written
- **log_etc_header()** writes a new escalation header record to the escalation transaction database table, mdsestrn
- **write_queue()** writes the (new) escalation message back to the bulk queue, and the global trans_complete flag is set to true.

Note that this differs from the standard skymds27 message handling. The idea with skymds27 is that messages are passed back, in different forms, to **process_contact()**, until successful delivery. Here the contact message, in its new form, is posted to the bulk queue directly by **write_queue()**. Possibly this is so that CDRs are generated by the message each time it passes through the bulk queue.

33.5.6 FUNCTION: PAGER_LOGIC

pager_logic() simply sends a pager message to the bulk queue. The thing to note though is that the pager message has a secondary history record attached (as a long string) as a way of transferring the secondary history record to the secondary queue.

- read and check the pager record in the msdpager database table, check that it exists and is active
- call **append_hist_to_msg()** to append the history record to the pager message.

The following comments appears in the code:

```
/* If pager number is a mobile phone then insert a quotation mark here */
/* because after this point we will append history record to the message */
/* which we don't want to insert quotation mark to any string of number */
```

and:

```
*****
/* The secondary record which will show who initiated this page is passed */
/* to skymds20 appended to the end of the message and logged after a */
/* primary hist record is written by skymds20. I know this sucks but none */
/* of us could think of a better way of passing the info to skymds20. */
/* ----- */
/* What the skymds20 needs is only the contact_id who initiated this page.*/
/* In the future when we have room in the queue structure we may store */
/* parent contact_id instead of trailing along the history record in the */
/* message. Phu Van Le: 01/07/1999 */
*****
```

- **queue_main()** posts the message to the bulk queue.
- set the global trans_complete flag to be true. This indicates that this contact has been delivered, and does not need any more processing.

33.5.7 FUNCTION: PROCESS_GRP AND GROUP_LOGIC

Groups are collections of message destinations, possibly of multiple types, all receiving the same message and treated , as far as the sender is concerned, as one single destination.

`process_grp()` coded in skymds27.c:

- gets an event number for the group pager message [why?]
- calls **`group_logic()`**

33.5.7.1 Function: group_logic

- uses **`load_groups()`** to loads the contact records for each member of the group into memory. If no group members are found then there is obviously a problem, so change this contact type to a “Supervisor Reminder” and return
- **`process_groups()`** is then called to send each individual message to the bulk queue. **`process_group()`** calls **`write_to_queue()`** once for each member of the group, passing the type of message / contact for that group member (Fax, pager message, reminder, escalation ...)

`write_to_queue()` formats and sends (with **`queue_main()`**) a message to the bulk queue.

33.5.8 FUNCTION: FAX_LOGIC

`fax_logic()`, coded in mds27fax.c, prepares a fax record for the fax server and sends it to the

- read the fax database table (mdsfax) to ensure that the fax record exists. If it doesn't there is obviously a problem, so change this contact type to a “Supervisor Reminder” and return.
- next the full_form_fax field of the database is tested. If this is meant to be a “full form” fax, the format is read from the forms database table (mdsforms) and the full-form-fax flag set to true
- The bulk queue record is prepared, including CDR details
- The fax message (contact) is sent to the bulk queue
- trans_complete is set to true, indicating that processing of this contact is complete.

33.5.9 FUNCTIONS: PROCESS_FRM AND FORMS_LOGIC

`process_frm()`, coded in skymds27.c,

- copies the message's (system) reference number to the escalations database table.
- calls **`forms_logic()`**

`forms_logic()`, coded in mds27frm.c, just strips out the contact message contained within the form message, and returns. As it does not set the trans_complete flag, **`process_contact()`** (the grandfather calling function) will process this contact.

There is a call to **`frm_register_pc<Company_Name>()`**, but, as far as can be seen, this is no longer used.

33.5.10 FUNCTION: TMS_LOGIC

TMS – telephone message service – stores a message from a caller for later retrieval by a client. The contact number and indial are stored on a phone record (mdsphone, defined in mdsphtrec.h)

- the phone record (necessary for a TMS) is found

33.5.11 FUNCTIONS: TXT_LOGIC AND PROCESS_TEXT

TMS: <Company_Name>'s Telephone Messaging Service collects messages from callers, the Call Centre Operators writing them to a text file stored in the <System_Name> system. `txt_logic()` compiles the stored messages collected by the Call Centre Operators into, and sends the filename and its contents to the city to which these messages should be delivered.

33.5.11.1 Function: process_text

- `build_indial_and_filename()` makes the correct filename from the email information supplied
- `build_text_str()` makes a text string from the email
- `write_message()` is used to write the email text to a file named as per the first step above. `write_message()`:
- uses `TXT_open_file()` (coded in mdstxrtn.c) to open the file
- the message is written to the file by `TXT_write_to_file()`, also in mdstxrtn.c
- Finally the email text message is stored on its “home” <System_Name> server, either directly or via the maintenance queue. A maintenance queue message is sent to the appropriate city with the file name *and* the text contained in the file. Maybe the storing of messages is done again (to a file with the same name) at the target server site, or maybe the maintenance queue appends the file’s contents to its maintenance queue message?

33.5.12 FUNCTION: REMINDER_LOGIC

Reminders are instructions to Call Centre Operators to do particular tasks, usually either call (or fax, or something) a customer, or report something to their supervisor.

- finds a record with the appropriate line number and contact ID in the phone table (mdsphone)
- call `generate_reminder()` to generate the reminder

33.5.12.1 Function: generate_reminder

`generate_reminder()` writes all kinds of information to the reminder record:

- standard Call Centre Operator instructions for the originating indial.
- contact type and ID are blanked out.
- the phone number is written.
- the greeting text is re-written, expanded by the `expand_msg()` function. Expansion is the process of replacing placeholders, like #NAM, with the correct and detailed text, like customers actual name.
- the machine ID, queue ID, time and status (unprocessed) are all added.
- the message itself, the indial number, contact ID and event number⁴⁶ are all added
- finally the history location and the pager number are added

- `REM_add_reminder()`, coded in mdsrem.c, adds the reminder to the reminder shared memory.
- A CDR record is constructed and written (with `log_cdr_rec_rem()`)

33.5.12.2 Function: REM_add_reminder

- finds where in reminder shared memory that the existence of this reminder should be written, and if there is a “reminder diversion” noted at that location `REM_add_reminder()` will dutifully update its choice of writing position.
- the reminders table (mdsrem) is locked and the reminder written
- The count of reminders is increased by one at the shared memory location found in the first step above
- we now sleep for one second, with the reminders table locked. Apparently this helps stop duplicate reminders
- unlock the reminders database table.
- write the database update (the addition to the reminders table) to the maintenance queue for replication to each of the other <System_Name> servers. skymds231, receiving the message from the maintenance queue at some other <System_Name> site knows to similarly update the reminder shared memory there also.

⁴⁶ Event numbers are escalation message resend identifiers. This reminder may have been an escalation in its (recent) former incarnation

33.5.13 FUNCTION: PHONE_LOGIC

phone_logic() writes a phone-someone reminder message to the reminder table. After copying some basic data (indial, country code, ...) **phn_generate_reminder()** sends the reminder. The handling of phone-reminders is very like ordinary reminders. Actually, why two separate functions?⁴⁷

33.5.13.1 phn_generate_reminder

- writes all kinds of information to the reminder record:

- machine ID
- date / time
- status (unprocessed)
- line number
- contact ID
- event number
- pager number
- a couple of history details

- Calls **REM_add_reminder()** to write the reminder to the reminder shared memory and database table. See a description of **REM_add_reminder()** in the previous section discussing **reminder_logic()**
- construct and post a CDR

33.5.14 FUNCTION: EMAIL_LOGIC

email_logic() processes customised form data that needs to be sent as emails to a client; it makes an email from a form. Once rewritten as an email message and posted back to the bulk queue the email is passes to skymds03 for normal delivery to the email gateway and beyond. **email_logic()** can also process contacts representing simple text (non-form) emails.

email_logic() is coded in mds27eml.c.

- first the email record for the stipulated line-number (email destination) is found in the email database table. If the record is not found a reminder is sent to the Call Centre supervisors.
- Next, if full_form_email is set to ‘Y’ in the email record, the form is retrieved from the forms table (mdsforms) and the raw data inserted into the form.

These formatted (text) emails are like the curses screens used in <System_Name>. Many full-form emails are generated by the “Forms” component of skymds01, the Call Centre Operator message input system, and thus many are simply reproductions of the original data-input form(s) displayed in the email. But the forms may also be of another matching type.

- further, for emails to be sent as laid-out forms various further details are added to the record
- The bulk queue ‘qu_details’ record is updated with info about this email about to be sent
- If the email is ‘full form’ the detailed full-form email-message is posted to the bulk queue, else the text from the contact (only) is just re-posted as a simple email.
- trans_complete is set to true. All done with this email contact.

⁴⁷ I don't know.

33.5.15 FUNCTION: SUPERVISOR_REMINDER

`supervisor_reminder()` formats and adds to the bulk queue reminders destined for Call Centre supervisors. Reminders of this type usually indicate some type of meta-problem with a message passing through <System_Name>, such as details of the message not being found where they should be, an escalated message has not been able to be delivered via any of its stipulated delivery mechanisms, or any other unexpected – but manageable – error has occurred

- first the contact type, error type, line number and event number of the message is extracted from the current (supervisor reminder) contact
- these are formatted into a text string, plus the words “(Inform Supervisor)”, which is copied into the global message buffer.
- the current date, time, location, and machine id are obtained
- a reminder message is now constructed from all this information, including the message in the global message buffer.
- If there is an event number, i.e. this is the result of an escalation, further information is read from the escalations table (mdsestrn), else information about this message is read from the bulk queue header. This is then written to the reminder.
- **`REM_add_reminder()`** is called to add the reminder to the appropriate reminder queue. See the **`reminder_logic()`** section for a description of **`REM_add_reminder()`**.

33.5.16 FUNCTION: ADV_LOGIC

`adv_logic()` merely calls **`add_client_stats()`**.

33.5.16.1 `add_client_stats`

coded in mdscstats.c, this function simply increments the message counter:

- for this pager / line number
- for this day of the week
- for this call type.

in the client stats database table (mdsclsts), in the home city of the indial only. In mdsclsts there is one record per line number, with 28 message counters: 7 (days of the week) x 4 message types.

A call is made to **`add_client_stats()`** with the line number, call type, and state. **`add_client_stats()`** then determines if the city that it is running in is the home city of the line number. If it is not, it writes a record to the maintenance queue and exits. Else it finds the record for this indial in the client stats file and increments the counter for the call type for this day of the week:

Code	Call type	Description
I	today: in	
O	today: out	
E	today: exc calls	Message taken via a contact (advanced processing)
F	today: fax	a fax has been sent

and writes the updated record back to the mdsclsts database table.

33.5.17 FUNCTION: CPA_LOGIC

Update the call patch table (mdscallp) with either a successful or unsuccessful value for the indial number specified by the call-patch contact message.

33.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

33.7 STOPPING SKYMDS27

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)

34 skymds06

34.1 SKYMDS06 OVERVIEW

skymds06 processes “timed events”, that is <System_Name> messages that need to be delivered (or not delivered) at or between certain times only. The list of these is something very similar to:

Timed Event	Description
Booked Calls	send a pager message at a set time
extra messages	(as per Booked Call)
delayed messages	(as per Booked Call)
follow-me start messages	(as per Booked Call)
follow-me finish messages	(as per Booked Call)
Reminders	Messages to the Call Centre Operators
Manual Reminders	Messages to the Call Centre Operators
Escalations	Messages that get new delivery if not acknowledged in time
Faxing Daily (et al) reports	faxing info to someone daily
emailing Daily (et al) reports	emailing info to someone daily
safety <Company_Name>	sending an alarm message if a customer hasn't called in (Safety <Company_Name>)
A-Party email report	send a report on A-Party (possibly bulk) emails or SMSs

Table 210: skymds06: Types of timed events

34.2 RUNNING SKYMDS06

Like all <System_Name> daemons, skymds06 is run from the start.mds start-up script. The operator, supervisor, or whoever is starting <System_Name> up is prompted with the question:

```
Start skymds11, skymds67, skymds18, skymds25, skymds28, skymds81,  
skymds06, skymds27, skymds10, remterm and skymds20 (y/n):
```

Presuming that the operator answers ‘Y’ skymds06 is run as per the command line below.

34.2.1 COMMAND LINE PARAMETERS

Parameter	Description	
-D	run in debug mode, listing detailed run-time information.	
-t <port>	the port to register skymds06 against	
-g <minutes>	grace time, how old timed events must be to be ignored	
-o <option list>	implement any (or all) of the following options [e.g. BtFM]	
	Option	Meaning
	B	Disable Booked Calls
	M	Disable Reminders (TMS)
	m	Disable Manual Reminders
	E	Disable Escalations
	F	Disable Fax Reports
	b	Disable Email Reports
	t	Disable Remote Data Delivery (Text)
	S	Disable Safety <Company_Name>

Table 211: skymds06: Command Line Parameters

34.2.2 EXAMPLE COMMAND-LINE

A command-line used to execute skymds06 at one <System_Name> site is:

```
$MDS_BIN_DIR/skymds06 -t /dev/dum00 -g 180 2>$LOG_SCR
```

indicating that skymds06 is run:

- registered against the port /dev/dum00 in the mdspports table
- timed events over two hours old (180 minutes) are ignored
- processing all types of timed events: no '-o' options used.
- with stderr redirected to LOG_SCR, #define'd in start.env to be /dev/tty0p5, a monitor somewhere.

34.3 PROGRAM DESCRIPTION

skymds06 processes all the <System_Name> messages that indicate a timed event, as per the list above. In its `process_timed_events()` loop the type of each timed event is determined and the appropriate handler function called.

34.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds06.

34.4.1 SOURCES FILES

Source File	Description
form_path.c	Make a file-path from an environment variable and a filename
gensubs.c	Contains various general-purpose routines that are separately compiled.
globvar.c	Functions to access the Global Variable shared memory used by multiple programs.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
mds06saf.c	Safety <Company_Name> Function.
mdscdr.c	Contains various general-purpose routines that are separately compiled.
mdscstats.c	Increment 'X-has-happened' count in a program statistics log file
mdsemqmain.c	Generic message email queuing routine.
mdsginit.c	A number of functions, which perform commonly, used initialisation code segments.
mdskbld.c	Contains functions to build and unbuild pager keys, and set-up a dummy pager.
mdslog.c	Generic error logging.
mdsphol.c	Functions to attach, check and modify Public Holiday shared memory. Used by multiple programs.
mdsportn.c	Routines associated with the ports file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurtn.c	Generic message queuing routine.
mdsrem.c	Functions to access the reminders queue, used by multiple programs
mdssemk.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssetup.c	Obtain set-up definitions from a text set-up file.
mdssubs.c	Contains various general purpose routines that are separately compiled
mdstmrtn.c	Contains functions, which perform operations to do with time.
skymds06.c	Timed event processing - booked calls- roster calls- escalations

Table 212: skymds06: Source Files

34.4.2 HEADER FILES

Header File	Description
emlparam.h	Email Parameter file
faxparam.h	Fax Report message/parameter buffer
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.: C time field definitions
globvar.h	Global shared memory structures
macros.h	User defined macros.: definitions & macros for common functions
mdsaprec.h	A-Party Master file
mdscdr.h	CDR definitions file
mdscdrec.h	CDR data structure
mdscsrec.h	Client stats master file
mdsctype.h	Used for validating message chars.
mdsderec.h	Ports security master file
mdsemrec.h	Email File (Online & Batch)
mdsflpsz.h	Standard <System_Name> includes.: field length & definitions
mdsfxrec.h	(FAX) Fax Master File
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsparam.h	parameters used by PMS system
mdsparec.h	Pager Master File
mdsphol.h	Public Holiday Shared Memory file
mdsphtrec.h	(TMS) Telephone Message Service
mdsporec.h	Ports security master file: Ports Record
mdsqrec.h	Contacts Message queue file: Destination Message queue file
mdsrem.h	Reminders shared memory definition
mdsrnrec.h	Reminders master file
mdsterec.h	Timed event file (incl. booked call)
saf/safflpsz.h	field length & definitions used by SAF
saf/saflgrec.h	Safety <Company_Name> log file
saf/safprrec.h	Safety <Company_Name> Profile
saf/safsprec.h	Safety <Company_Name> Service file

Table 213: skymds06: Header Files

34.4.3 DATABASE TABLES

Each of these header files contains (only) a C-struct that matched the structure of a database table used by skymds06.

Database Table	Header File	Description
mdsaparty	mdsaprec.h	A-Party Customer Info
mdscdrec	mdscdrec.h	Call details master
mdsclsts	mdscsrec.h	Client stats master
mdsdestm	mdsderec.h	Destinations master
mdsemail	mdsemrec.h	e-mail master
mdsfax	mdsfxrec.h	Fax master
mdsline	mdslirec.h	Indial number information
mdspager	mdsparec.h	Pager master
mdsphone	mdsphtrec.h	Phone master
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsqrec.h	Destination queues
mdsrem	mdsrnrec.h	Reminders queue
mdstimev	mdsterec.h	Timed events
saflg	saf/saflgrec.h	Safety <Company_Name> log file
safpr	saf/safprrec.h	Safety <Company_Name> Profile
safsp	saf/safsprec.h	Safety <Company_Name> Service file (table)

Table 214: skymds06: Database Tables

34.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description	Example
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide ...	O
MDS_DATA_DIR	The root of the data directories, queues, database and history.	/var/mds/data

Table 215: skymds06: Environment Variables

34.5 SKYMD06: MAJOR FUNCTIONS

34.5.1 FUNCTION: MAIN

- `initialise()` is called to set-up skymds06's run-time environment
- `process_timed_events()` undertakes the work of handling timed events in an infinite loop.
- `finalise()` is called once `process_timed_events()` exits to close all the database tables.

34.5.2 FUNCTION: INITIALISE

- read in the appropriate environment variables, MDS_MACHINE_ID and MDS_DATA_DIR
- parse the command-line
- use `mode_print()` to write to the log file a list of the types of timed event that have not been suppressed by the '-o' command line parameter.
- call `background()` to run skymds06 in the background.
- record that skymds06 has started – record in the mdsparts file
- obtain the shutdown semaphore, used to indicate when skymds06 should shut-down
- open all the database tables that we need
- open a file handle to the distribution queue from which skymds06 reads messages and the maintenance queue to which it writes them
- attach to the reminder, public holiday and global shared memory.

34.5.3 FUNCTION: PROCESS_TIMED_EVENTS

Running in an infinite loop `process_timed_events()` reads successive timed event messages / contacts from the timed events queue and processes these messages by passing each of them to the appropriate handler function.

In an infinite loop:

- tests the shut-down semaphore to see if it is time to terminate
- next it tests whether the machine on which it (skymds06) is running is a master machine. If not it sleeps. Thus the instances of skymds06 running on lnko (Melbourne non-master) and lnkx (Sydney non-master) do nothing until (say) the master machine (lnkm or lnks) crashes and these secondary machines are reassigned to 'master' status
- `check_for_work()` gets the next timed event. If nothing is found `process_timed_events()` sleeps before starting the loop again.
- `validate_pager_and_line()` is called to, as the name suggests, validate the pager and either validate or use the default line number (for pagers), or validate the line number (only) for non-pager indial services. However, if `validate_pager_and_line()` returns false then `process_timed_events()` carries on processing the current timed event. Some timed events (apparently) don't need a valid pager and/or line number.
- check that the timed event is still valid. Its 'release-date' must be younger than the 'grace time' passed in on the command line.
- Now determine the type of timed event, and pass control to the appropriate handler function

Code	Function Called	Description
TE_BOOKED_CALL	<code>process_booked_call()</code>	booked call – at a set time
TE_EXTRA_MESSAGE	<code>process_booked_call()</code>	extra message – at a set time
TE_DELAYED_MESSAGE	<code>process_booked_call()</code>	hold the message till a particular time
TE_FLME_START_MSG	<code>process_booked_call()</code>	new message destination: start message
TE_FLME_END_MSG	<code>process_booked_call()</code>	new message destination: end message
TE_TMSREM_MSG	<code>process_tms_reminder()</code>	
TE_MANUALREM	<code>process_manual_reminder()</code>	Manual reminder
TE_ESCALATION	<code>process_escalation()</code>	escalation – altered handling after t minutes
TE_FAX_REPORT	<code>process_fax_report()</code>	fax a periodic report
TE_EMAIL_REPORT	<code>process_email_report()</code>	email a periodic report
TE_SAFETY_<COMPANY_NAME>	<code>process_safety_<Company_Name>()</code>	‘time-out for client call-in’ alarm message
TE_AP_EMAIL_REPORT	<code>process_aparty_batch_reply()</code>	send report back to a Bulk SMS sender

Table 216: skymds06: Timed Event Handlers

- After the handler function returns, go around the loop again.

34.5.4 FUNCTION: PROCESS_BOOKED_CALL

- only process the booked calls that originated in the same city that ‘this’ server is running in
- don’t process business-day-only booked calls if it is not a business-day
- call `setup_queue_rec()` to prepare a bulk-queue message and `queue_main()` to post it
- call `update_timev_file()` to update the timed events file

Done.

34.5.5 FUNCTION: PROCESS_TMS_REMINDER

`process_tms_reminder()` prompts a Call Centre Operator to send the collected messages for a customer to that customer. Conditions apply.

- find the record (with the matching contact ID and line number) in the phone table
- read the record, delete it (so don’t use it twice), and send a delete message to the maintenance queue.
- if this reminder is not to be generated on a public holiday, call `public_holiday()` to test whether it is , in fact, a public holiday. Likewise weekends.
- Are there any messages to be delivered, and does the client actually want to know that there are (any messages)?
- construct the reminder
- call `REM_add_reminder()` to actually post the reminder to the reminder shared memory
- if posting the reminder was successful:
 - use `add_client_stats()` to record that a message has been sent to this line number
 - for some reason send an ‘undelivered message’ message to the bulk queue
 - construct and log a CDR
- call `update_timev_file()` to update the timed events file

34.5.6 FUNCTION: PROCESS_MANUAL_REMINDER

- if this manual reminder is not to be generated on a public holiday, call `public_holiday()` to test whether it is , in fact, a public holiday.
- construct the reminder
- call `REM_add_reminder()` to actually post the reminder to the reminder shared memory
- if posting the reminder was successful:
 - use `add_client_stats()` to record that a message has been sent to this line number
 - for some reason send a message to the bulk queue
 - construct and log a CDR
- call `update_timev_file()` to update the timed events file

34.5.7 FUNCTION: PROCESS_ESCALATION

- set-up a dummy pager record and write it to the bulk queue.
- call `update_timev_file()` to update the timed events file

34.5.8 FUNCTION: PROCESS_FAX_REPORT

- get the appropriate record from the fax table (mdsfax) to determine what type of fax to send.
- Construct the appropriate fax by copying information into a fax-parameter buffer

Fax Type Constant	Value	Description
FAX_LINE_REPORT	1	
FAX_PAGER_REPORT	2	
FAX_STARTEL_REPORT	3	Startel is an old, retired system that <Company_Name> at one time used to support <Company_1_Name> in Australia. This functionality has since been moved to an 'External Database' contact application, and will be moved again to <System_Name> II. Soon.
FAX_WORK_REQ_REPORT	4	Work Request Faxes are only generated by the <Company_3_Name> FRC applications

Table 217: skymds06: Fax Types

- now copy this fax-parameter buffer into a message and post it to the bulk queue
- call `update_timev_file()` to update the timed events file

34.5.9 FUNCTION: PROCESS_EMAIL_REPORT

Very similar to the process_fax_report above

- get the appropriate record from the email table (mdsemail) to determine what type of email to send.
- Construct the appropriate email by copying information into an email buffer

Email-Type Constant	Value	Description
EMAIL_LINE_REPORT	L	
EMAIL_PAGER_REPORT	P	
EMAIL_TEXT_REPORT	T	

Table 218: skymds06: Email Types

- use `setup_queue_rec()` to prepare a bulk queue record and `queue_main()` to post it to the bulk queue.
- call `update_timev_file()` to update the timed events file

34.5.10 FUNCTION: PROCESS_SAFETY_<COMPANY_NAME>

Safety <Company_Name> is a service to track employees working in dangerous environments and / or at night. The employee calls in, says what time they will be finished, and then are expected to call back at that time. If they are late an alarm is raised.

Database Table	Header File	Description
saflg	saf/saflgrec.h	Safety <Company_Name> log file
safpr	saf/safprrec.h	Safety <Company_Name> Profile
safsp	saf/safsprec.h	Safety <Company_Name> Service file (table)

Table 219: skymds06: Safety <Company_Name>: Database Tables Used.

- find the record in the safety <Company_Name> file with the same line number as that on the timed event
- Look in the Safety <Company_Name> log table (saflg). If the safety-<Company_Name> record has already been closed, then exit.
- Find the safety <Company_Name> record in the Safety <Company_Name> Service file (table).
- Find the safety <Company_Name> record in the Safety <Company_Name> Profile file (table).
- Now, one of three Safety <Company_Name> functions are called depending on the “step” of the Safety <Company_Name> process:

Step Code	Function Called	Purpose / Description
SAF_PAGE_STEP	SAF_page_msg()	page the worker. Do they respond?
SAF_PHONE_STEP	SAF_phone_msg()	phone the worker. Do they respond?
SAF_DURESS_STEP	SAF_duress_msg()	raise the alarm (!!). Worker not responding.

Table 220: skymds06: Safety <Company_Name>: Functions Called.

- Each of the SAF_* functions posts a message to the bulk queue, either a pager or a reminder for a call centre operator to activate.

If there is no response, the next step of the Safety <Company_Name> response, another timed event, will be activated.

- call **update_timev_file()** to update the timed events file

34.5.11 FUNCTION: PROCESS_APARTY_BATCH_REPLY

- Read the email table to get the email with the same line number and pager_contact_id as the (A-Party Batch Email Reply) timed event we are processing
- Delete it – and send the delete update to the other <System_Name> servers – so that it isn’t processed twice.
- Ignore the timed event if it is for a different city
- If the email type is EMAIL_2WSMS_REPLY_REPORT (that is: ‘h’) then fill in the ‘Pager Report Details’ version of an email parameter union (EML_PARAM – coded in emlparam.h) and copy it to a bulk queue message.
- Decide which email queue this email should go to. If the email is due to be sent in the future then send to the email queue “em”, else (immediate transmission) send to email queue “e2”.
- call **setup_queue_rec()** to prepare a bulk queue record and **queue_main()** to send the message
- call **update_timev_file()** to update the timed events file

34.5.12 FUNCTION: UPDATE_TIMEV_FILE

`update_timev_file()` reads the frequency field of the timed events record, and updates either the timed event itself (for minute and hour updates), or the timed events file for longer period timed events. Timed events fall into the following groups, and are handled the following ways:

Timed Event Type	Description
TE_FREQ_ONCE_ONLY	delete the current timed events database table record, and write a message to the maintenance queue to do likewise
TE_FREQ_ONCE_A_MINUTE	Use <code>TM_add_time_to_gmt_str()</code> to add a minute to the timed event's release date.
TE_FREQ_HOURLY: TE_FREQ_DAILY TE_FREQ_WEEKLY TE_FREQ_MONTHLY TE_FREQ_YEARLY	Use <code>TM_add_time_to_gmt_str()</code> to add an hour to the timed event's release date. 1. Use <code>calc_next_timed_event()</code> to add a day / week / month / year to the timed event's release date. 2. delete the current timed event from the mdstimev table 3. post the updated timed event to the maintenance queue to be reproduced in each city
TE_FREQ_BSWD:	1. Decide how many days to the next business day 2. use <code>calc_next_timed_event()</code> to add that many days to the timed event's release date 3. post the updated timed event to the maintenance queue to be reproduced in each city
TE_FREQ_VARIABLE:	1. read the (current) number of minutes to add from the timed event record. 2. use <code>TM_add_time_to_gmt_str()</code> to add this number of minutes to the timed event's current release_date

If the new release date is past the timed event's end date then delete the timed event and send a matching delete message to the maintenance queue to delete this event on the other <System_Name> servers.

34.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

34.7 STOPPING SKYMDS06

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)

35 skymds11r

35.1 SKYMDSD11R OVERVIEW

skymds11r receives data and messages from an external SMS Centre (SMSC) for addition to (and update of) the <System_Name> systems. skymds11r actually performs two tasks: either receiving messages from the SMSC **or** receiving SMS transmission data that <System_Name> uses to update history and billing data, the latter task is undertaken if skymds11r is run with the ‘-u’ command-line parameter. The lnks server in Sydney has 3 instances of skymds11r running, lnko in Melbourne one. The Adelaide server has an SMS transmission server (skymds11t) but no skymds11r.

Note that skymds11r has recently been renamed from (the former) skymds67a. Many references to the latter still abound.

35.2 RUNNING SKYMDSD11R

When running the <System_Name> start-up script start.mds, the user is prompted with a message like the following:

```
Start skymds11, skymds67, skymds18, skymds25, skymds28, skymds81,  
skymds06, skymds27, skymds10, remterm and skymds20 (y/n) :
```

presuming that they answer ‘y’, both skymds11t and skymds11r are run. Note the remaining reference to skymds67.

35.2.1 COMMAND LINE PARAMETERS

Parameter	Description
<u>Mandatory</u>	
-d <SMSC>	which SMSC to connect to.
-t <port>	the port to register this instance of skymds11r against in the mdsports database table.
<u>Optional</u>	
-D	run in debug mode, listing detailed run-time information.
-a <addresses>	address range
-u	only update the history and CDR files with info from the SMSC
-r	repair run
-B	‘Bureau Mode’: one way SMS messaging only
-C <code>	Carrier Code
-?	print skymds11r command-line parameters and exit

Table 221: skymds11r: Command Line parameters

35.2.2 EXAMPLE COMMAND-LINE

A command-lines used to execute skymds11r on lnks in Sydney are:

```
$MDS_BIN_DIR/skymds11r -D -B -CV -t /dev/dum55 -d SMSC_VODA 2>>$MDS_LOG_DIR/mds11rca1.log  
$MDS_BIN_DIR/skymds11r -D -u -t /dev/dum56 -d SMSC_VODA 2>>$MDS_LOG_DIR/mds11rhi.log
```

The first is run:

- in debug mode, listing detailed run-time information.
- in Bureau mode: no two-way paging
- going via the Carrier with the code ‘V’ – Vodafone
- registered against the port /dev/dum55 in the mdsports database table
- connecting to the gateway identified by SMSC_VODA
- with stderr (file descriptor 2) redirected to the file \$MDS_LOG_DIR/mds11rca1.log

The second is run

- in debug mode, listing detailed run-time information.
- only updating the history and CDR files. Not processing any other messages or database updates.
- registered against the port /dev/dum56 in the mdsparts database table
- connecting to the gateway identified by SMSC_VODA
- with stderr (file descriptor 2) redirected to the file \$MDS_LOG_DIR/mds11rhi.log

35.3 PROGRAM DESCRIPTION

skymds11r processes message received by <System_Name> from a particular SMSC, the choice of SMSC being passed to it on the command-line when it is run. skymds11r can perform two variations of this task:

- receiving data about previously sent SMS messages. This data is used to update (or re-write) the original history record of the SMS message with details of the carrier it has been sent over.
- receive actual messages from the SMSC

skymds11r's `main()` function passes control to either `process_sms_hist()` or to `process_smpp()` depending on whether the '-u' parameter was passed on the commandline. Both of these functions run until the shutdown semaphore reduces to zero or an unacceptable error occurs.

35.4 PROGRAM STRUCTURE

Following are listings of the main components that have gone into skymds11r.

35.4.1 SOURCES FILES

Source File	Description
skymds11r.c	This program. Receives SMS message responses from a SMSC.
alarmrtn.c	Generic alarm routine.
gensubs.c	Contains various general-purpose routines that are separately compiled.
globvar.c	Functions to access the Global Variable shared memory used by multiple programs.
isstart.c	Wrapper for C-ISAM isstart function - choose index and position the read-point
mdscdr.c	Contains various general-purpose routines that are separately compiled.
mdscomio.c	Various serial IO routines
mdscomm.c	Socket io and Serial module.
mdsemqmain.c	Generic message email queuing routine.
mdsginit.c	A number of functions, which perform commonly, used initialisation code segments.
mdshisio.c	Generic history file handling routines.
mdskbld.c	Contains functions to build and unbuild pager keys, and set-up a dummy pager.
mdslog.c	Generic error logging.
mdsportn.c	Routines associated with the port file.
mdsqmain.c	Generic message queuing routine.
mdsqmnt.c	Queues file updates for transmission to other machines.
mdsqurtn.c	Generic message queuing routine.
mdssemk.c	Check the nominated semaphore and return TRUE if it has dropped.
mdssmppa.c	communications functions related to communications to a SMSC using the SMPP protocol.
mdstmrtn.c	Contains functions, which perform operations to do with time.

Table 222: skymds11r: Source Files

35.4.2 HEADER FILES

Header File	Description
gentypes.h	User defined types and booleans
gettime.h	C time field definitions.
macros.h	User defined macros.: definitions & macros for common functions
mds2wr.h	Structure of the 2-way-messaging reply-storage database table
mdsaprec.h	A-Party Master file
mdscdr.h	CDR definitions file
mdscdrec.h	CDR data structure
mdsderec.h	Ports security master file
mdsflsz.h	Standard <System_Name> includes.: field length & definitions
mdshirec.h	History primary recs file: New History logging file
mdshsrec.h	Secondary History file (audits)
mdslirec.h	Needed for the acdmem.h LINEMBX struct: History primary file
mdsncrec.h	Network Connection master file
mdsparam.h	parameters used by PMS system
mdsparec.h	Pager Master File
mdsporec.h	Ports security master file.
mdsqrec.h	Destination Message queue file
mdsshrec.h	The structure of the SMS message delivery status database table
mdssmpa.h	SMPP definitions.

Table 223: skymds11r: Header Files

35.4.3 DATABASE TABLES

Each of these header files contain (only) a C-struct that matched the structure of a database table used by skymds11r.

Database Table	Header File	Description
mdsaparty	mdsaprec.h	A-Party Customer Info
mdscdrec	mdscdrec.h	Call details master
mdsdestm	mdsderec.h	Destinations master
hiccyymmdd	mdshirec.h	the 'logical' history
hsyymmdd	mdshsrec.h	History secondary storage
mdsline	mdslirec.h	Indial number information
mdsneten	mdsncrec.h	Network connection
mdspager	mdsparec.h	Pager master
mdsports	mdsporec.h	Ports security
mdsqxx.txt	mdsqrec.h	Destination queues
mdspager	mdsshrec.h	SMS delivery status

Table 224: skymds11r: Database Tables

35.4.4 REFERENCED ENVIRONMENT VARIABLES

Environment Variable	Description	Example
MDS_BIN_DIR	data directories, queues, database and history.	/var/mds/data
MDS_MACHINE_ID	M,O Melbourne S,X Sydney A Adelaide ...	O
PASSWORD	password to access the SMSC	8ashgxca87c6acigak
SERVICE_TYPE		,
SOURCE_ADDR		921050
SOURCE_ADDR_NPI		3
SOURCE_ADDR_TON		2
SUBSCRIBER_SUB_TYPE		
SYSTEM_ID	<Company_Name>	
SYSTEM_TYPE	<Company_Name>comm	

Table 225: skymds11r: Environment Variables Used

35.5 SKYMDS11R: MAJOR FUNCTIONS

35.5.1 FUNCTION: MAIN

As mentioned above, skymds11r performs one of two tasks: receiving messages *or* receiving data about messages. The choice as to which road to follow is decided based on receipt (or otherwise) of a ‘-u’ command-line parameter, ‘-u’ indicating deal with the transmission data, otherwise receive SMS messages themselves.

- call `initialise()` to prepare skymds11r’s run-time environment
- if ‘-u’ was passed on the command-line pass control to `process_sms_hist()` to receive and update transmission data about the SMS messages previously sent to the SMSC
- otherwise call `process_smpp()` to receive SMS messages

both of the process functions run as infinite loops and do not return until skymds11r terminates.

35.5.2 FUNCTION: INITIALISE

The `initialise()` function sets up skymds11r’s run-time environment, via the following steps:

- reads in the required environment variables, 10 or so.
- parse the command-line
- call `background()` to run skymds11r as a daemon
- catch a few signals
- determine the sleep-time
- get the shut-down semaphore used to flag that skymds11r should terminate
- open and prepare the maintenance queue. All the history updates go via the maintenance queue.
- call `HIST_start_session()`
- `GEN_get_netw_connect_rec()` gets the appropriate network connection record.
- `setup_smsc_dests()` reads all the records in the mdsdests table with the protocol field set to ‘S’ into the smsc_dests array.
- use `PORT_update_port_rec()` to record skymds11r against the port passed in at the command-line (via the ‘-t’ parameter)
- if we are reading transmission data to update the <System_Name> history and CDR file, then use `get_carrier_smsranges()`.
- `initialise_tlv_options()`, coded in mdssmppa.c, is now used to allocate space for the TLV variables. TLV variables are used in the transmission of SMS messages.

35.5.3 FUNCTION: PROCESS_SMS_HIST

`process_sms_hist()`, receives the transmission data back from the SMSC and files it appropriately in the <System_Name> database system. The function `final_state()` does a lot of the work.

In an infinite loop:

- check the shutdown semaphore. Is it time to halt?
- read the next record from the SMS delivery status database table (mdssmsdel). This is the store of all the messages that have been sent, but are still awaiting data back from the SMSC for them to be completed.
- copy the source address, phone number, and message into a SMPP_PACKET record
- call `final_state()` to integrate this data with the history database table.
- if it was all-OK then delete the record from delivery status database table (mdssmsdel).

35.5.4 FUNCTION: FINAL_STATE

`final_state()` updates the history server with data about the call.

- check that, with an international code, the phone number starts with ‘61’; that we haven’t picked up some stray calls from Russia, lets say
- convert the phone number to a pager number and validate it.
- call `HIST_net_open_session()` to open the appropriate history files for this SMS call. They may be in a different city, so this call works transparently over a network.
- find the original record of this call in the history table.

This is tried up to 100 times before `final_state` gives up. `NHIST_read_prim()` is actually used to get the record. Now, as there may be a delay in the original record being written to the history in the first place, if the record is not found in the history tables use `write_final_state_tmp_hist()` to store the record for later processing. So why try 100 times then, huh?

- secondary history. Use `NHIST_posn_sec()` and `NHIST_read_sec_next()` to read the matching secondary history record
- Now write the updates history record to the maintenance queue so that it is written to the history files both locally and at the other <System_Name> sites. Some manipulation and copying is done and the message is written to the maintenance queue.
- finally `HIST_net_close_session()` is called, to close the history files being read.

35.5.5 FUNCTION: PROCESS_SMPP

`process_smpp()` runs in a double infinite loop, the outer loop maintaining and re-establishing connections to the SMSC, the inner loop reading and storing messages.

The outer loop – connecting to an SMSC.

`process_smpp()` receives an SMS message (in SMPP protocol) from an SMSC.

- first, if there has been a change in how we are connected to the SMSC, use `PORT_update_port_rec()` to note any new connection details
- call `connect_and_bind_to_smsc()` to, as the name suggests, connects and binds (sends a bind packet) to the SMSC.

`connect_and_bind_to_smsc()` calls `init_tcpip_connection()` or `init_x25_connection()` to open a socket connection to the SMSC. If `connect_and_bind_to_smsc()` fails, we have to switch to another telecommunications carrier’s SMSC. The next few commands deal with this.

- if `connect_and_bind_to_smsc()` has failed `GEN_get_next_netw_connect()` is tried and we start the outer loop again
- if that fails `GEN_get_netw_connect()` is used. If that fails its all over and we break-out

The inner-loop – reading a message from the SMSC

- `smpp_recv()` first is used to get the next SMPP communication (SMS message) from the SMSC
- if no message is received from the SMSC, then we call `send_enquire_<Company_Name>()` to test that the connection. If the connection is down, then we break to the outer loop to re-establish the connection. If the packet is one that we don’t expect then print an error and sleep for two seconds
- send a reply to the SMSC with `reply_to_smsc()`
- finally, if we are not connected to our server of choice then this is the time to change.

End of outer loop:

Back to maintaining the network connections. If we have got to here we must have broken out of the inner loop with either some sort of network problem or program termination, so we close the network connections and re-open them at the start of the next loop... or maybe never.

- `close_connection_smse()` well, closes the connection to the SMSC
- again check the shutdown semaphore. If it has been set to 0, then it is time to die.

35.5.6 FUNCTION: CONNECT_AND_BIND_TO_SMSC

`connect_and_bind_to_smse()` sets up a TCP/IP connection to the SMSC.

- first the TCP/IP binding structure is prepared.
- then in an infinite loop either set-up an X.25 or a TCP/IP connection depending of the flags set
- now send a test message to the SMSC via `mess_to_smse()` and test the response.
- If the repines was OK, then all done → exit. If it was not OK, the connection is closed and we go around again.

This function continues looping until a connection is made.

35.5.7 FUNCTION: CHECK_ENQUIRE_<COMPANY_NAME>

`check_enquire_<Company Name>()` is a short function to test the status of the connection to skymds11t's SMSC.

- `smpp_recv()` reads a packet from the SMCC. timeouts are treated as success
- other than timeouts, anything other than SMPP_E_OK (message OK, value 0x00) is treated as an error.
- we now test the value of `smpp_enquire_<Company Name>_resp()` when it sends a message to the SMSC.
- again, anything other than SMPP_E_OK is an error.

35.5.8 FUNCTION: SMPP_RECV

Get a packet from the SMSC, with the option of waiting for / expecting a particular type of packet. This function loops through NUMBER_OF_ATTEMPTS times, currently 3.

- First set the timeout for various type of packet requests. An alarm is set to go off at that time.
- Now do the read.
- First the code for handling failures. Basically, if the read timed-out start the loop again. Any other error then break out and exit the function.
- read the received packet, get the info we need from it, including its type
- Now handle the situation of receiving an unexpected type of packet. Different combinations of expected –vs– actual packet type trigger the responses of either exiting the function or ignoring the error.
- Check that the stated length of the packet is correct and that it was all received.
- check that, given there were no other errors, the packets own status is SMPP_E_OK.
- finally call `check_fields()` to run a few checks for different types of packets.

35.5.9 FUNCTION: REPLY_TO_SMSC

`reply_to_smsc()`, as the name suggests, sends a reply back to the SMS.

- if we need to send an error, write the error codes to the packet.
- most replies will pass to `process_delivery_sm()` to have their details appropriately recorded in the <System_Name> database
- if not being processed by `process_delivery_sm()` else configure the response to an Are-You-There enquiry
- else handle invalid packet types, by writing out an error but also sending an ACK acknowledge signal.
- call `smpp_send()` to send the packet

35.5.10 FUNCTION: PROCESS_DELIVERY_SM

`process_delivery_sm()` records the details of a SMS message in the <System_Name> database and appropriately prepares both a response packet and response database table record based on the SMS packet received.

- do some initial, standard formatting of the response SMS packet
- for Return Receipt messages (ESM_CLASS_DELIVERY_RECEIPT) and in Bureau Mode (one-way-only) then just call `final_state()` to store the message data in the history table.

for all other types of messages

- check that the return phone number is in the correct format
- determine (with `get_sms_source_type()`) what type of SMS this was:
 - One way / Normal
 - A-Party Email originated
 - A-Party Bulk email originated
 - A-Party SMPP originated. Came from another carrier, into <System_Name>, and we sent it out again.
- Prepare a response (if not a one-way message). This involves populating a receiver reply database table record (detailed in mds2wr.h) and writing it to the 2 way reply messages store database table (mds2wr).
- finally set the response packet command status to 'OK' (OK = SMPP_E_OK = 0x00).

35.6 LOG MESSAGES PRODUCED

There are three types of log messages produced by <System_Name> programs:

- Those sent through the <System_Name> log handling functions
- Those always written directly to stderr
- Those only written to stderr while in debug mode.

The log messages produced by skymds11r are listed following:

35.7 STOPPING SKYMDS11R

1. Set the shutdown_semid to non-zero.
2. Find its PID and KILL -9 it (as root)

Coding Reference Section

[Application Independent]

36 Curses Overview

In <System_Name>, as in most teenage Unix applications, the “curses” system is used to make each and every GUI component, every dialog and window that appearing on a terminal screen. “curses” does this utilising only:

- existing characters from the ASCII character set,
- character attributes
- the curses screen management library.

36.1 SOME IMPORTANT PARTS OF A CURSES SYSTEM

Here we find described a few of the (many) important structures in a curses application, so you get the rough idea. The curses functions operate on one or more of these components. Everything is displayed as one or more terminal characters, though lines (of ‘-‘ or ‘+’ or whatever) are treated as separate objects.

36.1.1 CHARACTERS

There are two components to a curses character, the digit and its attribute:

- attr_t The attribute. An integral type that can contain at least an unsigned short. = the attribute
- chtype A curses character, an ASCII digit + an attribute ‘attr_t’.

So, using attributes, characters, lines and boxes can be underlined, reversed, in reverse video, or whatever.

Characters may be manipulated in this way. For example the call: addch('*' | A_REVERSE) adds a ***** character .

36.1.2 WINDOW

An opaque window representation. The Curses functions permit manipulation of window objects, which can be thought of as two-dimensional arrays of characters and their renditions. A default window called stdscr , which is the size of the terminal screen, is supplied. Others may be created with newwin().

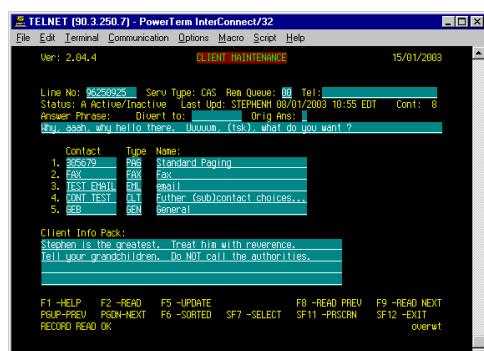
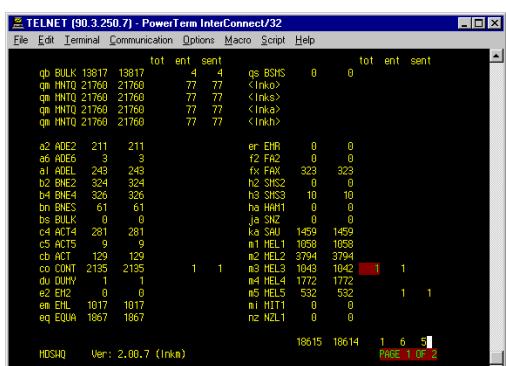
Variables declared as WINDOW * refer to windows (and to subwindows, derived windows, and pads)

36.1.3 SUBWINDOWS

A subwindow is a window, created within another window (called the parent window), and positioned relative to the parent window.

36.1.4 SCREEN

An opaque terminal representation.



36.1.5 TERMINAL

A **terminal** is the logical input and output device through which character-based applications interact with the user. A TERMINAL is an opaque data type associated with a ‘terminal’. The TERMINAL data structure contains information about the capabilities of the terminal (as defined by terminfo), terminal modes, and the current state for input and output operations. Each SCREEN is associated with a TERMINAL.

36.2 PREPARING THE CURSES SYSTEM

First, your program must include the curses header file `<curses.h>`. This defines everything your (curses) program will ever want to know, including

- video attributes,
- line drawing characters,
- input values use names, such as `A_REVERSE` and `ACS_HLINE`.

Next, according to a curses man page, “the application must call `initscr()` or `newterm()` before calling any of the other functions that deal with windows and screens”. These functions transfer terminal information from a termcap entry or an environment variable to the curses program.

More specifically, to get character-at-a-time input without echoing (most interactive, screen-oriented programs want this), the following sequence should be used:

<code>initscr()</code>	initialise the curses system
<code>cbreak()</code>	pass characters straight through to the curses interface
<code>noecho()</code>	don't write the characters onto the screen (let curses manage that)

Most programs would additionally use the sequence:

<code>nonl()</code>	Enable processing of the 'Enter' key
<code>intrflush(stdscr, FALSE)</code>	Don't flush buffers every time the application is interrupted
<code>keypad(stdscr, TRUE)</code>	Convert function key presses

36.3 CURSES FUNCTIONS: A BRIEF OVERVIEW

36.3.1 MOVEMENT

There are two movement functions in the curses suite

- `move(x,y)`, and
- `wmove(x,y)`

Of course many output functions update the cursor position themselves. Note that the upper left-hand corner of a window has the coordinates (0,0), not (1,1).

36.3.2 VARIATIONS, FORMS OF CURSES FUNCTIONS

As curses is a text-based system, a large proportion of the functions available manipulate characters or strings in one way or another. These functions, regardless of their category, are usually available in the following four versions:

1. simple: do the task. e.g. `addch(ch)`
2. move: move to (x,y) then do the task. e.g. `maddch(x, y, ch)`
3. operate in a particular window. e.g. `waddch(win_id, ch)`
4. move in a specific window, then do the task, e.g. `wmvaddch(win_id, x, y, ch)`

For a list of curses functions in the major function groups, see the function list at the end of this curses overview.

36.3.3 CLASSES OF CURSES FUNCTIONS.

Again, for a list of curses functions in the major function groups, see the function list at the end of this curses overview.

characters: output

Add a character to screen at current position
Add the character and then refresh the user's screen.

int addch(chtype ch)
int echochar(ch)

strings of characters: output

Write an arbitrary length string at the current cursor location

int addstr(char *str);

all characters: attribute manipulation

turn attributes off (in the selected window)
turn attributes on (in the selected window)
set the attributes in the selected window

int attroff(int attributes)
int attron(int attributes)
int attrset(int attributes)

characters: Input

int getch(void) read a character
void echo(void) do show the input to the user. [Bad idea. Let curses manage that stuff]
void noecho(void) don't show the input [Good idea]

characters: Input with processing

void nocbreak(void) character/character input, can do special processing.
void raw(void) do keep formatting
void noraw(void) don't keep formatting
int keypad(stdscr, TRUE) translate function key presses
int keypad(stdscr, FALSE) pass function key presses onto the underlying program

strings of characters: input

void cbreak(void) line input

wait for input?

void nodelay(void) ERROR is no character
int halfdelay(int tenths) wait 'tenths' 1/10ths of a second for a character

clear the screen

int erase(void); put blanks everywhere (will be written when refresh is called)
int clear(void); put blanks and let terminal be cleared next time

36.3.4 IMPLEMENTING SCREEN UPDATES

So after one has made any changes to a screen, call refresh(). That's it. After a call to refresh, the WINDOW or TERMINAL will be updated with all the changes since the previous call to refresh().

36.3.5 FINALISING CURSES AND COMPILING

36.3.5.1 program termination

Before you the program exits, make sure it calls endwin().

36.3.5.2 compile time

At compile time you must <Company_Name> in the curses library: cc [flag ...] file ... -lcurses

36.4 MORE INFO ON CURSES

1. The curses manual and specification is on the <Company_Name> Communications G:\ drive in **G:\technology\Product Systems Development\System_Name\1\Books\Curses_manual.pdf**
2. The HP-UX 11.00 installation that <System_Name> is developed on has only two man pages concerning curses:
man curses, a description of the curses.h header file
man curses_intro, about the curses system
Other Unix systems do have much more extensive curses documentation.

36.5 CURSES FUNCTION LIST

These are not, actually, all the possible curses functions, but will do for now.

36.5.1.1 Add (Overwrite) Functions

Functions available	Description	Example
[mv][w]addch()	add a character	addch()
[mv][w]addch[n].str()	add a character string	addchstr()
[mv][w]add[n].str()	add a string	addstr()
[mv][w]add[n].wstr()	add a wide character string	addnwstr()
[mv][w]add_wch()	add a wide character and rendition	add_wch()
[mv][w]add_wch[n].str()	add an array of wide characters and renditions	add_wchnstr()

Table 226: curses: Add (Overwrite) Functions

36.5.1.2 Change Rendition Functions

Functions avialiable	Description	Example
[mv][w]chgat()	change renditions of characters in a window	chgat()

Table 227: curses: change rendition functions

36.5.1.3 Deletion Functions

Functions avialiable	Description	Example
[mv][w]delch()	delete a character	delch()

Table 228: curses: deletion functions

36.5.1.4 Input from Keyboard to Window

Functions avialiable	Description	Example
[mv][w]getch()	get a character	getch()
[mv][w]get[n].str()	get a character string	getnstr()
[mv][w]get_wch()	get a wide character	get_wch()
[mv][w]get[n].wstr()	get an array of wide characters and key codes	get_wstr()

Table 229: curses: input functions

36.5.1.5 Explicit Cursor Movement

Functions avialiable	Description	Example
[w]move()	move the cursor	move()

Table 230: curses: movement functions

36.5.1.6 Read Back from Window

Functions avialiable	Description	Example
[mv][w]inch()	input a character	inch()
[mv][w]inch[n]str()	input an array of characters and attributes	inchnstr()
[mv][w]in[n]str()	input a string	innstr()
[mv][w]in[n]wstr()	input a string of wide characters	innwstr()
[mv][w]in_wch()	input a wide character and rendition	in_wch()
[mv][w]in_wch[n]str()	input an array of wide characters and renditions	inwchnstr()

Table 231: curses: read-from-window functions

36.5.1.7 Insert Functions

Functions avialiable	Description	Example
[mv][w]insch()	insert a character	insch()
[mv][w]ins[n]str()	insert a character string	insnstr()
[mv][w]ins[n]wstr()	insert a wide-character string	ins_nwstr()
[mv][w]ins_wch()	insert a wide character	ins_wch()

Table 232: curses: insert functions

36.5.1.8 Print and Scan Functions

Functions avialiable	Description	Example
[mv][w]printw()	print formatted output	printw()
[mv][w]scanw()	convert formatted output	scanw()

Table 233: curses: print functions

36.5.1.9 Function Keys

Key	Key codes
Break key	KEY_BREAK
The four arrow keys	KEY_DOWN, KEY_UP, KEY_LEFT, KEY_RIGHT
Home key	KEY_HOME
Backspace	KEY_BACKSPACE
64 Potential function keys	KEY_F(n)
Delete line	KEY_DL
Insert line	KEY_IL
Delete character	KEY_DC
Insert char or enter insert mode	KEY_IC
Exit insert char mode	KEY_EIC
Clear screen	KEY_CLEAR
Next page	KEY_NPAGE
Previous page	KEY_PPAGE
Enter or send	KEY_ENTER

Table 234: curses: function keys

37 <System_Name> curses utilities

The <System_Name> specialised curses functions are collected in the source-files ‘editscrn.c’ and ‘utils.h’, as discussed below, and have been written to provide ease to the developers and a consistent look-and-feel to the resulting users of the dialogs, windows, and screens.

And the functions are ...

37.1 FUNCTIONS IN EDITSCRN.C

Function	Description
edit_scrn()	Manages interactions where a user can, and does, manipulate a screen's structure.
edit_report_scrn()	Similar: customised for the manipulation and editing of report screens
mnt_get_input()	Reads input from the user
mnt_get_date_time()	Produces the correctly formatted date and time.
mnt_vali_date()	Validate a supplied date.
display_scrn()	Show the screen
mnt_get_invis_string()	Read in, but don't echo a string. For passwords.

Table 235: curses: <System_Name> custom functions

in editscrn.h:

Structure	Description
struct _editscrn;	Despite the name, this structure holds one single component of a screen, a data entry field. Arrays of these _editscrn structures are passed into: <ul style="list-style-type: none">- edit_scrn()- edit_report_scrn(), and- display_scrn() where they are each read and placed appropriately on the screen. The <u>array</u> of _editscrn structures is the <System_Name> screen object, holding the form, edit order, and field properties of the screen.

Table 236: curses: <System_Name> screen structure

37.1.1 FUNCTION: EDIT_SCRN

OK. edit_scrn():

1. receives, as a function parameter, a pointer to an array (i.e. an array) of _editscrn structures that, as just noted, comprise a <System_Name> screen. This screen must have been displayed by the calling function and thus currently visible to the user.
2. collects the user's keystroke input,
3. interprets it, and
4. edits both:
 - the screen-display being watched by the user, and
 - the array of _editscrn structures (i.e. the stored representation of the screen) accordingly.

Now, in even more detail (can you bear it?):

- the “current character” is set to ASCII 0, i.e. NULL

The remainder of the function is a large, infinite loop – until someone presses escape. On each pass a single key-press is processed, the current field possibly being edited, adjusted, moved or bypassed, all driven by the user's keyboard, maybe typing text, possibly tabbing between fields or jumping between selections.

- skip any un-editable display-only screens
- process a couple of key presses from the previous loop, the ‘Enter’ key (add a line) and the help key (display any help available)

Now do default processing on whatever the current field is.

- take a copy of the field’s current contents.
- Now, testing for some attributes of the current field, turn these attributes on the “overlay” window, which defines the window in which the fields must be contained.
- if the field is a date / time field strip out any leading zero’s.
- if the field is a password field, call `mnt_get_invis_string()` to read the password from the user
- if none of the above, read the next key-press from the user
- Again_testing for some attributes of the current field, turn these attributes on the “overlay” window.

The remainder of `edit_scrn()`, still inside the infinite loop, is a switch statement, moving the current field to another field dependent on the key-press.

37.1.2 FUNCTION: EDIT_REPORT_SCREEN

This is a clone of `edit_scrn()`, very much the same code and the same structure. I’ll leave this one as an exercise for the reader.

37.1.3 FUNCTION: MNT_GET_INPUT

The function processes the text the user enters in an editable field.

- prepare and display any existing text.
- move the cursors editing cursor to the position of the next letter after the end of the text string.

Now process each character entered. First, how are we going to handle another character?

- Is it a valid character?
- Is there room?
- Now, if we added the character in the middle of the string move the other characters one to the right to accommodate the insertion.
- Have we entered enough text that we have to scroll?

And now, what does this character do?

- | | |
|-----------------------|---|
| • Ctrl-C pushed? | Clear the field. |
| • ‘Enter’ pushed? | If we are at the end of the message or on a blank line, this means that its time to exit.
Otherwise just add a new line as usual, so draw a screen and scroll the text up. |
| • Escape? | Exit with no update. |
| • Arrows & Backspace. | their usual function |
| • Insert Key | use <code>disp_ins_over_str()</code> to toggle the Insert / Overwrite status. |
| • A Function key? | Beep!! ... and discard it |

OK, all done. So now time to cleanup.

- Strip all trailing spaces off any string we have obtained
- return the last character we processed back to the calling function.

37.1.4 THE OTHER FUNCTIONS

37.1.4.1 display_screen

Yes, the display the screen. This function is just one big ‘for()’ loop that positions and displays each _editscrn in the array of them (_editscrn structures) that define the screen.

37.1.4.2 mnt_get_date_time

A complex, fully-featured date / time function that really belongs in a library, not lost amongst this code.

Among the parameters, one can supply:

- The initial date (before processing)
- Limits – to trap unacceptable dates
- a format string, and
- where to display it.

You get the idea. The code in editscrn.c if you actually do want to pour over its internals yourself.

37.1.4.3 mnt_vali_date

A date-validation function. Is this date possible, probable, and with-in bounds.

37.1.4.4 mnt_get_invis_string

As described before, read-in a string with-out echoing it on the screen.

37.2 UTILITY CURSES FUNCTION IN UTILS.H

Function	Description
clear_win_region()	Writes '' over a region of a window
create_popup_win()	As per the name, displays a pre-constructed curses window structure
delete_popup_win()	Undo the above
confirm_exit()	Ask “Are you sure you wish to clear the message (y/n)?” upon exiting.
edit_or_send()	Prompt to either re-edit or send the message
confirm_delete()	confirm that the user does want to delete the record chosen
zero_pad()	left pad a number with zeros if it needs to be a particular length
popup_message()	Display a message box to the user and wait for them to hit ESC
display_error()	Display an error to the user and wait for them to hit ESC
disp_info_str()	Display a message box <u>including date, time</u> to the user, and wait for them to hit ESC
disp_ins_over_str()	Present the insert / overwrite option to the user
getkey()	use getch() to obtain the next valid key press.
wait_on_key()	wait for some input
get_invis_string()	for reading in (hidden) passwords, and the like
answer_yn()	force the operator to enter “Y” or “N”
get_string	Allows an operator to enter a string of characters into a window field.
get_numeric()	Allows an operator to enter a string of numerics into a window field.
_get_input()	A long, detailed function to read user input. Used by get_string() and get_numeric().
validate_char()	Ensure that a character is valid.
draw_string()	write characters to the screen
insert_status()	Updates the INSERT / OVERWRITE status
pop_wait_msg()	presents a simple ‘please wait’ message.
rev_select()	return a user’s selection from a presented menu.
select_item()	highlight the item on the list just selected
erase()	erase an item from a curses menu
disp_oper_mod_time()	Display the time
get_relevant_tzone()	Force the operator to select what time zone they are in.
popup_report_email_win()	popup an email address input window for report generation
disp_report_fax_win()	Fill in display items on fax window

38 Examples from a <System_Name> curses application

38.1.1 DECLARE FUNDAMENTAL ENTITIES

First an application will declare the screen structures that it needs.

```
char blank_line[BLANK_LINE_SIZE+1],  
  
WINDOW *titlewin = (WINDOW *)NULL,  
        *mainwin = (WINDOW *)NULL,  
        *fkwins = (WINDOW *)NULL,  
        *statuswin = (WINDOW *)NULL;
```

38.1.2 STRUCTURE OF A SCREEN

As noted above the `_editscrn` struct holds the screen widgets, their size, position, type and variable to which they are bound.

```
typedef char LINE_NUM[LINE_NUM_SIZE];  
typedef char ANSWER_PHRASE[ANSWER_PHRASE_SIZE];  
  
LINE_NUM      sc_line_num;  
ANSWER_PHRASE sc_answer_phrase;  
QUEUE_ID      sc_queue_id;  
LINE_NUM      sc_divert_line_num;  
char          sc_status;  
TEL_NBR       sc_phone_num;  
char          sc_orig_ans_flag;  
  
struct _editscrn scm_template[] =  
{  
    {sc_line_num,      LINE_NUM_SIZE,      0, 9, " ~", 0,0, 24, 3,0},  
    {sc_queue_id,     QUEUE_ID_SIZE,     0,48, " ~", 0,0, 0, 3,0},  
    {sc_phone_num,    TEL_NBR_SIZE,      0,56, " ~", 0,0, 0, 4,0},  
    {sc_divert_line_num, LINE_NUM_SIZE, 2,30, " ~", 0,0, 0, 5,0},  
    {&sc_orig_ans_flag, sizeof(char),   2,52, " YN", 0,0, 0, 5,0},  
    {sc_answer_phrase, ANSWER_PHRASE_SIZE, 3, 0, " ~", 0,0, 0, 6,0},  
  
    {(char *)NULL,     CONTACT_ID_SIZE,   6, 5, " ~", 0,0, 5, 9,0},  
    {(char *)NULL,     CONTACT_TYPE_SIZE, 6,17, " ~", 0,0, 5,10,0}  
}
```

38.1.3 PREPARE THE SCREEN

Usually a function along the lines of `what_am_I_set_screen_array()` sets up the interactions and fine details of the curses screen where, for example, each text-field's next (tab and arrow) and previous (tab and arrow) text-field are set.

```
for (i=0; i<12; i++)  
{  
    adv_desc[i].help_mode = 0;  
    adv_desc[i].display_only = 0;  
    adv_desc[i].display_attr = 0;  
}  
  
adv_desc[0].up_field = 11;  
adv_desc[1].up_field = 0;
```

```

:
:
adv_desc[3].down_field = 4;
adv_desc[4].down_field = 5;
adv_desc[5].down_field = 6;

```

38.1.4 MAKE THE SCREEN APPEAR

`create_popup_window()` writes the already constructed curses window structure to the user's terminal screen.

```

if ((advwin = create_popup_win(mainwin,19,80,2,0,&saveadv,&save_y,&save_x,
                               WIN_NOFLAGS)) == (WINDOW *)NULL)
{
    sprintf(err_line,"Failed to create advisory window\n");
    mds_log(TRUE);
    delete_popup_win(titlewin,headwin,savehead,save_y,save_x,WIN_REFRESH);
    beep();
    return;
}

```

38.1.5 PROCESS THE USER'S INTERACTIONS

Each curses application will have a function that manages the user interaction. Here, within the UI function, the `edit_scrn()` function is used to read the user's keystrokes, moving the insertion point in response to tab and arrow movements, and passing function key presses (e.g. F2 and F3 in this example) back here to trigger the calling of appropriate handler functions.

```

for (;;)
{
    ret_key = edit_scrn(adv_desc, mess_prompt, advwin, &edit_field,
                        ADV_SCRN_SIZE, DISP_INFO_INS_OV_RHS);

    if (ret_key == KEY_F(2)) { /* F2 */
        adv_file_read(advwin,stswin);
        edit_field=0;

    } else if (ret_key == KEY_F(3)) { /* F3 */
        adv_file_add(advwin,stswin);
        edit_field=0;

    } else if (ret_key == ... ...

```

39 Bulk SMS Specification

NB: This appendix is a duplication of the document “**SMS Services: Implementation Specification**”

39.1 OVERVIEW

One of the key requirements of the new services is to make them accessible to as many customers as possible, with minimal to no additional expenditure to their network or systems. With this in mind, it has been proposed that Email be used as the transport mechanism for delivering information into <Company_Name> for the purpose of SMS delivery.

The new services, including the Email and SMS gateway, will be built onto <System_Name> utilising existing infrastructure and messaging capabilities.

39.2 SMS FUNCTIONAL REQUIREMENTS

39.2.1 PAGING VIA EMAIL

This feature will be built and hosted on <System_Name>, allowing customers to send a message to a predefined pager number via an email message. The extraction of the pager number, message text and real time delivery will be fully automated.

39.2.2 'B' PARTY SMS VIA EMAIL

This feature will be almost identical to Paging via Email, with the addition of SMS specific security checking, complying with individual carrier requirements. The subscriber must be registered on both Sky<Company_Name> and <System_Name>. The extraction of the phone number, message text and real time delivery will be fully automated.

39.2.3 'A' PARTY SMS VIA EMAIL

This service will introduce phase one of ‘A’ party messaging. It requires the sender of the Email to be registered on both Sky<Company_Name> and <System_Name>, before they are able to send SMS messages. The ID (phone number) does not need to be registered with <Company_Name>, and billing CDR records are captured against the ‘A’ party account. The extraction of the sender, phone number, message text and real time delivery will be fully automated.

39.2.4 'A" PARTY BULK SMS

This service will allow for both generic or individual messages to be sent to multiple phones via a store and forward method. It requires the sender of the Email to be registered on both Sky<Company_Name> and <System_Name>, before they are able to send SMS messages.

The required information will be provided in a text file attachment, utilising various tags to clearly identify the data components. The ID’s (phone numbers) do not need to be registered with <Company_Name>, and billing CDR records are captured against the ‘A’ party account.

Priority levels will also be supported within this service, determining the order in which messages are delivered. The extraction of the sender, phone numbers, messages and real time delivery will be fully automated.

39.3 SYSTEM ARCHITECTURE / MODULES

The new products / services listed earlier will form part of a new <System_Name> Mail Gateway Processing module. It will become part of, and integrate into the existing <System_Name> environment, but itself be broken down into various logical components.

Figure 1: “Email Messaging Diagram” below is a high level systems diagram listing the main components:

[imported graphic removed]

39.3.1 CORPORATE MAIL GATEWAY

<Company_Name>'s corporate mail gateway(s) will be configured to recognise several new domains listed later in this document. The new domains will allow the corporate gateway to route the emails to different <System_Name> servers, both for redundancy and load sharing purposes.

Once routed to a <System_Name> server, it will become the responsibility of the <System_Name> system to process the email.

39.3.2 SENDMAIL

sendmail is the Mail Transport Agent currently being used by <System_Name> for the purpose of receiving and sending Email messages. sendmail is a very powerful and complex protocol which can be configured to perform very specific tasks. The options and features of sendmail are outside of the scope of this document.

The sendmail utility on each <System_Name> server will be configured to identify the following domains:

- | | |
|--|---|
| pager.<Company_Name>.co
m.au. | This is an existing domain used for RF paging, which will also be used for 'B' party SMS. The maximum size of the email will be set to 2MB. |
| sms.<Company_Name>.com
.au. | This will be a new domain used solely for 'A' party SMS, where the sender must be registered with <Company_Name>. The maximum size of the email will be set to 2MB. |
| bulksms.<Company_Name>
.com.au. | This will be a new domain used solely for 'A' party bulk SMS. The maximum size of the email will be set to 2MB. |

When an email for either of the above mentioned domains is received by sendmail, it will perform the following tasks:

- Initiate the <System_Name> Email Gateway application (mdsemgw), passing the entire contents of the email to the gateway application via 'stdin'.
- Extract the sender's email address and pass it to the gateway application via a command line argument.
- Extract the recipient's email address and pass it to the gateway application via a command line argument.

39.3.3 <System_Name> EMAIL GATEWAY

The <System_Name> Email Gateway application will serve as the doorway to the <System_Name> messaging system. Its primary purpose will be to receive emails from the sendmail agent, and provide first level security and authentication processing. It will also be responsible for identifying potential abuse of the services provided, i.e. dumping, and take appropriate action in protecting the network, systems and service levels.

39.3.3.1 Initialization

Due to the gateway acting as a relay application, it is initiated every time a message is forwarded to it by sendmail.

During initialisation, the application will first extract both the sender and recipient of the email passed to it as command line arguments by sendmail. It will also attach itself to a Global Shared Memory Block, used to keep track and history of incoming emails and potential dumping.

Before accepting the contents of the email, the domain portion of the sender's email address is extracted and used to verify whether that domain or sender has been either permanently or temporarily blocked from sending messages. This is done by accessing the Email Blocking table defined in Appendix A.1.

To ensure the checking for a blocked user is performed as efficiently as possible, the following data access rules will apply:

1. A "Greater than or Equal to" I/O is done using the domain portion of the index only. If a record is not found, or a different domain is returned, or a system error occurs, it is assumed the sender is not blocked.
2. If the record returned has the same domain but a blank sender, this indicates that the entire domain is blocked, including the current sender.
3. If the record returned has the same domain and the same sender name as the sender of the email, then that sender has been blocked.
4. If the record returned has the same domain, but a different name, then an exact read is done using both the domain and sender name. If the record is found, then the sender is currently blocked.

If the sender is identified as being blocked, which was set automatically by the gateway application, and has been blocked for more than a pre-defined period of time i.e. 1 hour, then the blocking record is deleted. The sender is now unblocked and normal process can continue.

If the sender has been manually blocked by the Maintenance application, or still within the automatically blocked period, the email will not be processed. If the notified_sender field on the Email Blocking record is set to 'N', then this indicates that we have not yet notified the sender that they have been blocked. In this situation, an email is sent to the sender with an error message stating that they have been blocked, and that their email messages will not be processed. The notified_sender field is then set to 'Y', avoiding multiple messages being sent back to the sender, that may, in extreme circumstances, cause congestion across our network. The gateway application will exit with a return code of zero '0', notifying sendmail that it has successfully completed its task.

39.3.3.2 Automatic Blocking

In order to protect our systems and network against deliberate and accidental abuse, an automatic blocking feature will be implemented into the gateway application.

Via a circular array stored in shared memory, the gateway application will keep track of the last 'x' number of emails received ie. 1000, grouped by service type ('A' Party SMS, Bulk SMS, etc), including timestamp. This array will be used to monitor certain thresholds which senders must comply to. The suggested thresholds per sender are:

pager.<Company_Name>.com.au	50 per minute, averaged over 2 minutes
sms.<Company_Name>.com.au	50 per minute, averaged over 2 minutes
bulksms.<Company_Name>.com.au	1 per 5 minute

If the gateway application identifies a particular user to have exceeded the threshold, an automatic blocking record is created, and an email sent back to the sender, informing them that they have been blocked for a certain period of time. Any further emails received by the same sender, will be ignored until the blocking period has elapsed.

39.3.3.3 Processing of Email

Once the gateway application has completed its blocking and email dumping checks, it is ready to start processing the incoming email. To keep this step as quick and efficient as possible, no special logic or validation of the email contents is done by the gateway application.

The email is received via the gateway application's stdin, and is written in its entirety to disk as a unique ASCII text file. The text file name must be unique, and will be generated via the following method:

A date and time string with the following format CCYYMMDDHHMMSS is generated using the system clock, represented in GMT.

A rotating sequential number is stored and used within the Global Shared Memory block, ranging from 1 through to 99999. This is extracted and incremented within the shared memory block, and then appended to the date string. This will then form the unique file name which can cater for 99,999 emails per second.

i.e. 20010511123545.00034

The text file will be written into a specific directory under the general <System_Name> data directory, dependant on the service (domain) the email is sent to.

pager.<Company_Name>.com.au	/var/mds/data/email/pager
sms.<Company_Name>.com.au	/var/mds/data/email/sms
bulksms.<Company_Name>.com.au	/var/mds/data/email/bulksms

For each text file created, an entry is written into the Email Queue file (mdsemqu), setting a trigger for the next application to take over and process the contents of the email. The format of the Email Queue file is defined in Appendix A.2. The Email Queue file is 3 logical files, merged into the one physical file.

39.3.3.4 Gateway Log File

As with all messaging applications, it is important to be able to monitor and analyse all events encountered by the email gateway and associated applications. For this reason, a generic text email log file will be used for the purpose of logging all events and exceptions encountered whilst processing any of the email services. The log file will include such things as:

- Basic information of each incoming email such as Timestamp, Sender and Recipient.
- Information regarding an auto blocking / unblocking of a user.
- Emails ignored due to a user being blocked.
- Reply Emails
- Etc.

One log file will exist per day, and will reside in the /var/mds/data/email/logs directory. The name format of the log file is as follows:

emgwYYMMDD.log

where

YY = Year

MM= Month

DD = Day

39.3.4 REPLY EMAIL

There are several conditions which may result in a Reply Email being generated. These include the <System_Name> Email Gateway, as well as individual modules which process the contents of the email. In any event, the reply email will have the same format, with a unique and descriptive message.

Sender: Online_Admin
Subject: Message Failed

Body:

*Subscriber Id: <pager/mobile>
Date: dd/mm/ccyy hh:mm:ss tzn
Status:<one of the following messages>
Subscriber Id does not exist or is currently inactive
Email messaging not enabled for requested subscriber
Your account has not been registered for this service
Your account is currently inactive
You have been temporarily blocked from this service due to excessive use. Any subsequent emails will be ignored
You have been permanently blocked from this service. Any subsequent emails will be ignored
Bulk SMS Email contains more than one attachment and cannot be processed.
Bulk SMS Email attachment does not conform to agreed specification.
Bulk SMS attachment content error – Line# nnnnnn
Invalid PIN.
Invalid gateway.
Batch Number/Pin is invalid format.
Message successfully despatched to <COMPANY NAME> messaging system.
You have no registered account for Escalation or Roster service.
Your Escalation/Roster account is currently inactive.
Your account has no permission for this Escalation/Roster.
Email message not processed due to message containing no data.*

39.3.5 EMAIL BLOCKING MAINTENANCE

Other than automatic blocking, which is only in force for a short period of time, there is also a mechanism whereby a user can be permanently blocked. This can be done via a new Email Blocking Maintenance function available via the Files menu on <System_Name>. This function will access the Email Blocking table referenced in Appendix A.1.

The maintenance function will contain the following screen information:

Domain Name
Sender Name
Blocking Commencement Date
Blocking Type
Sender Notified Indicator

The Domain Name and Sender Name fields are the only two enterable fields on the screen. The Domain name is mandatory, but the Sender name is optional. If the Domain Name is entered without a Sender Name, then all senders from that domain are blocked. If a Sender name is entered, then that sender only is blocked from that domain. Multiple senders from the same domain can be blocked.

The Blocking Commencement Date is display only, and is used to determine when the blocking record was created.

The Blocking Type field is display only, and is used to determine who was the initiator of the blocking record, either manually via the maintenance function, or automatically via the Email Gateway application. If a blocking record created automatically is updated via the maintenance function, its Blocking Type is converted to manual, and cannot be automatically unblocked by the Email Gateway application.

The Sender Notified Indicator field is display only, and is used to determine whether a reply email has been sent to the blocked user. When a user is manually blocked, this field is set to 'N'. If the user sends an email, the Email Gateway

application will identify the user as blocked, but not yet notified. A reply email is generated, and the field set to ‘Y’, to avoid multiple replies being sent back to the user for every email they send.

39.3.6 ‘B’ PARTY PAGING / SMS

All ‘B’ Party Paging / SMS will be processed by module *mdsem01* which will be a daemon process running permanently in the background. This process will periodically check the Email Queue file as defined in Appendix A.2, extracting the oldest record first.

If a record is found, the subscriber id (pager or mobile phone number) is extracted from the *recipient_address* field of the record, and validated against the <System_Name> database. If the id does not exist, or is not active, a reply email is sent to the original sender, and the email text file deleted along with the Email Queue record. An entry is also written to the log file for future reference if required.

If the id exists and is active, its *Subscriber_type* properties are checked against the <System_Name> database to determine whether it’s a third party SMS service, ie. Optus SurePage, etc. If so, the standard <System_Name> security module is used to determine whether the subscriber has security for this type of messaging via Email. This is very important as subscribers of Optus SurePage can not be sent any messages other than those generated from the contact centre, as a result of a call diversion. If the id belongs to a third party SMS service, and does not have security for this type of message generation, a reply email is sent back to the original sender, and the email text file deleted along with the Email Queue record.

Once the id and security validations are complete and successful, the email text file can then be processed. Using a set of generic system libraries defined in Appendix B.1, the email subject, body text and attachment names are extracted from the email text file, created by the Email Gateway application. The data structures used to store the extracted information are referenced in Appendix B.2.

The text message is constructed using the email subject and body text, along with the names of any attachments represented in the following format: <>. The contents of any attachments will not be processed. Control characters and multiple spaces are removed from the message string, ensuring only printable characters are delivered. If the email contains no subject, no message and no attachments, the word “Blank Message” is defaulted as the message string.

As an example, if an email was received with a subject of “*Attention Required*”, body text of “*Please review attached document and provide feedback*” and an attached Word document called “*price_list*”, the following message string will be constructed:

“*Attention Required Please review attached document and provide feedback <>*”

The length of the message string to be transmitted will be determined by the *max_msg_length* field stored against the id on the <System_Name> database, set by the provisioning system. During this phase of the service, truncation will occur if the constructed message string is longer than the *max_msg_length* variable.

Once the message string is constructed, it is queued into the generic <System_Name> Bulk Queue file, where it will be processed and delivered accordingly. Both the Email Queue record and email text file are deleted from the system.

At this point, the ‘B’ Party Paging / SMS module will check the Email Queue file for any additional records and repeat the processes until all records are processed. If there are no further records to be processed, the module will sleep for a pre-determined period of time (extracted from the port record), and then check the Email Queue file again, repeating the process over and over again.

39.3.7 ‘A’ PARTY SMS

As detailed previously in this document, there are two types of ‘A’ Party SMS functions, Real Time and Bulk. In order for a customer to use either function, they must first be registered on both the Sky<Company_Name> and <System_Name> systems.

39.3.7.1 Provisioning

All ‘A’ Party customers will be provisioned and maintained on the Sky<Company_Name> system, which in turn will update the <System_Name> system via a new transaction type in the existing interface protocol. <System_Name> will store all the ‘A’ Party customer information in a new ‘A’ Party table defined in Appendix A.4. This will reside on all <System_Name> servers, and will be kept up to date real time as new records are added, changed or deleted by Sky<Company_Name>.

39.3.7.2 Authentication

Once an Email is received on either of the ‘A’ Party domains, it must first be validated before any further processing can occur. There are three possible levels of authentication, with the first and only mandatory one being the Sender of the Email. This is extracted from the Email header, and validated against the ‘A’ Party table in the following order:

All characters are converted to uppercase, and an exact read is done on both the sender and domain names. If an exact match is found, then the ‘A’ Party has at least been registered and the first level of authentication complete.

If an exact match is not found, then a “Greater than or Equal to” I/O is done on the ‘A’ Party table with only the domain name portion of the index populated. If a record is not found, or a different domain is returned, then the sender is not registered for these features.

If the record returned has the same domain, but a blank sender, then the entire domain has access the these features.

If the record returned has the same domain but a different sender, then the current sender of the Email is not registered for these features.

If the sender of the Email is not registered, then a reply Email is sent along with a descriptive error message informing them that they are not registered for this service. The Email will not be processed any further, and the Email text file deleted along with the Email Queue record. Once the ‘A’ Party has been authenticated, there are two additional security checks which can be executed, provided they have been enabled against the ‘A’ Party record. These include the Email Gateway Domain, and a PIN.

If the *Gateway_Domain* field is populated, the Email header is scanned, checking whether the Email has been routed through the nominated gateway. If the details are not found in the Email header, then a reply Email is sent to the originator of the Email informing them that their nominated gateway is invalid.

If the *PIN* field is populated, it is validated against the expected PIN embedded within the mobile phone number (recipient Email address). If the PIN is not found or is incorrect, a reply Email is sent to the originator of the Email informing them that their nominated PIN is invalid.

PIN’s for both Real Time and Bulk ‘A’ Party SMS are embedded within the recipients Email address as follows:

Real Time ‘A’ Party SMS

mobile_number-XXXX@sms.<Company_Name>.com.au
Where mobile_number = Mobile phone number in local format
 XXXX = PIN

Bulk ‘A’ Party SMS

batch_number-XXXX@bulksms.<Company_Name>.com.au
Where batch_number = Customer nominated batch number (max 8 char)
 XXXX = PIN

If the above mentioned security checks are successful, then the ‘A’ Party Email can be processed.

39.3.7.3 Real Time Processing

Using a set of generic system libraries defined in Appendix B.1, the email subject, body text and attachment names are extracted from the email text file, created by the Email Gateway application. The data structures used to store the extracted information are referenced in Appendix B.2.

The text message is constructed using the email subject and body text, along with the names of any attachments represented in the following format: <>filename.extension>>. The contents of any attachments will not be processed. Control characters and multiple spaces are removed from the message string, ensuring only printable characters are delivered. If the email contains no subject, no message and no attachments, the word “Blank Message” is defaulted as the message string.

The length of the message string to be transmitted will be fixed at 160 characters. During this phase of the service, truncation will occur if the constructed message string is longer than the 160 characters.

Once the message string is constructed, it is queued into the generic <System_Name> Bulk Queue file, where it will be processed and delivered accordingly. Both the Email Queue record and email text file are deleted from the system.

At this point, the Real Time ‘A’ Party SMS module (mdsem02) will check the Email Queue file for any additional records and repeat the processes until all records are processed. If there are no further records to be processed, the module will sleep for a pre-determined period of time (extracted from the port record), and then check the Email Queue file again, repeating the process over and over again.

39.3.7.4 Bulk Processing

Using a set of generic system libraries defined in Appendix B.1, the email attachment is extracted from the Email body. There can only be one attachment, and it must be an ASCII text file. If this not the case, a reply Email is sent to the originator of the Email, along with a descriptive message informing them of the error, and the Email text file deleted along with the Email Queue record.

The text file attachment is first parsed and validated for accuracy and completeness. The attachment is also checked to determine how many messages it will generate, and if above the threshold of 10,000, it will not be processed. All data tags and validation rules are covered later in this document. Based on the contents of the text file attachment, messages are constructed and queued into one of four separate queue records, depending on the nominated priority.

Considering the number of messages that can be processed for a single attachment, it is important to keep track of where in the attachment we have processed up to. In the event the process is terminated and restarted, it will be able to continue from where it was last up to, rather than starting at the beginning of the attachment, and re-sending messages more than once. This is done by keeping a checkpoint against the Email Queue record, of where in the attachment file we have processed up to.

The length of the message string to be transmitted will be fixed at 160 characters. During this phase of the service, truncation will occur if the constructed message string is longer than the 160 characters.

At this point, the Bulk ‘A’ Party SMS module (mdsem03) will check the Email Queue file for any additional records and repeat the processes until all records are processed. If there are no further records to be processed, the module will sleep for a pre-determined period of time (extracted from the port record), and then check the Email Queue file again, repeating the process over and over again.

39.3.7.5 Format Of Text File Attachment

There are four data tags which can be used within the bulk file attachment. Each one can be used multiple times within the same attachment,

<#BULKMSG#>	This is a general Bulk message that will be sent to a group of phones. Message text can be up to 160 characters long, and can not contain any non printable characters. Message text can be on the same line as the tag, or a line on its own.
<#PRIORITY#>	If not specified, will default to the value specified on the 'A' Party database. Values are 1 through 4 with 1 being the highest priority. Can be on the same line as the tag, or a line on its own.
<#BULKID#>	Contains all the mobile phone numbers, which will receive the defined <#BULKMSG#> message. All numbers must be between 8 and 10 digits in length, and separated by a Carriage Return, Line Feed or both. The first mobile number can be on the same line as the tag, or a line on its own.
<#UNIQUEID#>	Allows a unique message to be sent to a single phone. The first value is the mobile phone number, and the remainder is considered to be the message text. Same message text rules apply as for <#BULKMSG#>. Both the phone and message can appear on the same line as the tag, together on a separate line, or each on a separate line.

As soon as a tag is repeated, it supersedes its previous value.

The following example show how the tags can be used. The same results will be achieved with all the different examples.

39.4 EXAMPLES

39.4.1 EXAMPLE 1

```
<#UNIQUEID#> 0411123456 BATCH NUMBER 445637 COMMENCING
<#BULKMSG#> 4 BEDROOM HOME JUST LISTED - PLS CALL ABC REAL ESTATE
<#PRIORITY#> 2
<#BULKID#> 0411399483
0412334562
0455876223
0411410444
<#PRIORITY#> 1
<#BULKMSG#> AUCTION THIS SATURDAY FOR 24 SMITH ST HAS BEEN CANCELED
<#BULKID#>
0477885377
0411334564
0422334093
<#UNIQUEID#> 0411123456 BATCH NUMBER 445637 COMPLETED
```

39.4.2 EXAMPLE 2

```
<#UNIQUEID#>
0411123456 BATCH NUMBER 445637 COMMENCING
<#BULKMSG#>
4 BEDROOM HOME JUST LISTED - PLS CALL ABC REAL ESTATE
<#PRIORITY#> 2
<#BULKID#>
0411399483
0412334562
0455876223
0411410444
<#PRIORITY#>
1
```

```
<#BULKMSG#>
AUCTION THIS SATURDAY FOR 24 SMITH ST HAS BEEN CANCELED
<#BULKID#>
0477885377
0411334564
0422334093
<#UNIQUEID#>
0411123456
BATCH NUMBER 445637 COMPLETED
```

39.4.3 EXAMPLE 3

```
<#UNIQUEID#>
0411123456
BATCH NUMBER 445637 COMMENCING

<#BULKMSG#>
4 BEDROOM HOME JUST LISTED - PLS
CALL ABC REAL ESTATE

<#PRIORITY#> 2

<#BULKID#>
0411399483
0412334562
0455876223
0411410444

<#PRIORITY#> 1

<#BULKMSG#>
AUCTION THIS SATURDAY FOR 24 SMITH ST HAS BEEN CANCELED

<#BULKID#>
0477885377
0411334564
0422334093

<#UNIQUEID#>
0411123456
BATCH NUMBER 445637 COMPLETED
```

39.4.3.1 Priority Queue processing

The priority queue processing module (mdssms01) will be responsible for extracting ‘A’ Party Bulk SMS messages from the four different priority queue files, and inserting them into the generic <System_Name> Bulk Queue file, where it will be processed and delivered accordingly

The module will always start with priority queue 1 until all records are processed. Once done, it will then process up to 5 records from the other 3 queues, starting with priority 2 and working towards the priority 4 queue. After 5 records are successfully processed, irrespective of which queues they came out of, it will revert back to the priority 1 queue and repeat the process over and over again.

39.4.3.2 Capture of CDR's

All messages generated and queued by the ‘A’ Party SMS modules, will generate a unique CDR record, including the ‘A’ Party ID and Batch Number for Bulk SMS.

No history or stats records will be captured as we do not have the mobile phone numbers registered on our Sky<Company_Name> or <System_Name> systems.

Email Blocking Table Definition

Field Name	Type	Description
Domain_name	Char (60)	The domain portion of an email. <u>Mandatory</u> .
Sender_name	Char (60)	The Email senders name. <u>Optional</u> .
Initiator		If left blank, all senders for the nominated domain are blocked. Identifies who initiated the blocking. <u>Mandatory</u> ‘M’ = Manual ‘A’ = Automatic
Notified_sender	Char (1)	Determines if the sender has already been notified by email that they have been blocked. Y/N
Blocked_date_time	Char (14)	The date and time the user was blocked. Automatically set by the system.
Operator	Char (8)	The user or application which created / updated the blocking record. Automatically set by the system.
Mod_date_time	Char (14)	Date and time the record was last modified. Automatically set by the system.
INDEX Unique: domain_name, sender_name		

Table 237: Appendix 1 – SMS Definition: Email Blocking Table Definition

Email Queue Table Definition

Field Name	Type	Description
Type		Identifies the type of Email that has been queued. ‘P’ = ‘B’ Party Paging and SMS ‘A’ = ‘A’ Party SMS ‘B’ = Bulk SMS
Queue_date	Char (14)	The date and time the Email was received and queued to be processed. Stored in GMT.
File_path_name	Char (80)	Fully qualified text file path and name.
File_checkpoint	Char (8)	Processed position pointer in text file.
Sender_address	Char (80)	The full email address of the sender. Used for reply email if required.
Recipient_address	Char (80)	The address the email was destined for. Used for reply email if required, and ID identification.
INDEX Unique: type, queue_date		

Table 238: Appendix 1 – SMS Definition: Email Queue Definition

A-Party Provisioning Transaction Format

Note: All fields are mandatory unless specifically indicated as optional.

Field Name	Type	Description
STX	Char (1)	Start Of Text character
Sequ_nbr	Char (1)	Sequence Number 1-9.
Message_id	Char (7)	Message identifier 1-9999999.
Rec_len	Char (3)	Length of record
Subscriber_type	Char (1)	G = SAGRN, L = <Company_Name>
Tran_type	Char (2)	AA = Add, AC = Change, AD = Delete
Domain_name	Char (60)	Domain portion of Email Address.
Sender_name	Char (60)	Sender portion of Email Address. <u>Optional</u>
Gateway_domain	Char (60)	The gateway the Email should have been received from. <u>Optional</u> .
A_party_id	Char (8)	'A' Party identifier. Numeric with leading zeros
Client_num	Char (8)	Client / Account number
PIN	Char (4)	Pin Code used for security validation. <i>Optional</i>
Name	Char (25)	Name / Description of 'A' Party. <i>Optional</i> .
Priority_flag	Char (1)	1-4. 1 = Highest, 4 = Lowest
Status	Char (1)	A = Active S = Suspended I = Inactive
Remote_contact_operator	Char (8)	Remote Escalation / Roster Operator code
Remote_contact_indial	Char (10)	Remote Escalation / Roster Indial number
ETX	Char (1)	End Of Text character
CRC	Char (4)	Cyclic Redundancy Check
CR	Char (1)	Carriage Return Character
XON	Char (1)	Xon character

Table 239: Appendix 1 – SMS Definition: A-Party Provisioning record format

A-Party Table Definition

INDEX:1 Unique domain_name, sender_name		
Field Name	Type	Description
Domain_name	Char (60)	The Domain portion of the 'A' Party Email Address.
Sender_name	Char (60)	The Email Senders name. Optional. If left blank, all senders for the nominated domain are assumed.
Gateway_domain	Char (60)	The gateway the Email should have been received from. Used for additional security validation.
A_party_id	Char (8)	Identifier of an 'A' Party. This can relate to one or many senders.
Client_num	Char (8)	Billable client number.
PIN	Char (4)	Pin Code used for security validation
Subscriber_type	Char (1)	G = SAGRN, L = <Company_Name>
Normalised_name	Char (25)	Descriptive name for the 'A' party.
Priority_flag	Char (1)	1 = Highest, 4 = Lowest
Status	Char (1)	A = Active S = Suspended I = Inactive
Remote_contact_operator	Char (8)	Remote Escalation / Roster Operator Code
Remote_contact_indial	Char (10)	Remote Escalation / Roster Indial number
Mod_operator	Char (8)	User who last modified the record.
Mod_date_time	Char (14)	Date / Time record was last modified. GMT

INDEX: Unique: domain_name, sender_name

Table 240: Appendix 1 – SMS Definition: A-Party record format

39.5 SPECIAL SMS FUNCTIONS

The following are generic libraries that will be written to perform various functions against the raw contents of an email. Data structures and definitions used by these functions are referenced in Appendix B.2.

FILE *MIME_parse_file (path, filename, inmessage)

Initialise the appropriate data structure *inmessage* and open up the physical email data file, returning the logical unit number.

void MIME_init_message(inmessage)

Initialise the data structure setting strings to NULLs and integers to zero. Called by `MIME_parse_file()`.

int MIME_extract_data (infile, inmessage)

Extracts individual components of the email file referenced by *infile*, and set component reference pointers within the data structure *inmessage*.

int MIME_get_basicpart (inmessage, out_buff, max_len, extract_subject)

Get content message (called *basicpart*) referenced by *inmessage* structure and store into *out_buff* up to *max_len* characters. If *extract_subject* is set to true, then include the email subject within *out_buff*, otherwise ignore it.

char *MIME_get_ret_receipt_addr(inmessage)

Extract the return receipt address to determine if a delivery acknowledgement is required.

int MIME_check_for_attachments(inmessage)

Determine whether the email has any attachments or not, excluding the *basicpart*. Returns True or False

int MIME_check_att_type(inmessage, att_nbr, att_type)

Check whether the requested attachment number, excluding the *basicpart*, is of a particular type. Returns True or False.

int MIME_cleanup (inmessage)

Delete all extracted components referenced within the data structure.

int MIME_sendmail (fromaddr, toaddr, ccaddr, subject, message, fname1, fname2, ..., NULL)

Encode message and filenames attached into the email and send it to "toaddr".

39.6 BASIC BULK-SMS DATA DEFINITIONS

```
#define MIME_CONTENT_NONE          0
#define MIME_CONTENT_TEXT           1
#define MIME_CONTENT_AUDIO          2
#define MIME_CONTENT_IMAGE          3
#define MIME_CONTENT_VIDEO          4
#define MIME_CONTENT_APPLICATION    5
#define MIME_CONTENT_MULTIPART     6
#define MIME_CONTENT_MESSAGEPART   7
#define MIME_CONTENT_UUDECODE       8

#define MIME_MAX_FILES              10

#define ENC_NONE                    0      /* Encoding none */
#define ENC_QP                      1      /* Quoted-printable */
#define ENC_BASE64                  2      /* Base 64 */

typedef struct mime_header
{
    EMAIL_ADDRESS from_addr;
    EMAIL_ADDRESS to_addr;
    EMAIL_ADDRESS ret_receipt_addr;
    EMAIL_ADDRESS subject;
} mime_header_t;

#define MIME_HEADER_SIZE sizeof(mime_header_t)

typedef struct mime_body
{
    FILE_PATH att_filename;           /* File path include file name */
    long filesize;                  /* Size of the file */
    int content_type;               /* Type of content in sub-message */
    int encoding_type;              /* Coding type of attachment, QP, Base64, None */
} mime_body_t;

#define MIME_BODY_SIZE sizeof(mime_body_t)

typedef struct mime_message
{
    mime_header_t header;
    int content_type;               /* Content type of the message */
    int num_files_att;              /* Number of files attached, incl basicpart */
    mime_body_t bodypart[MIME_MAX_FILES];
} mime_message_t;

#define MIME_MESSAGE_SIZE sizeof(mime_message_t)
```

40 <System_Name> Coding Notes

40.1 EACH PROGRAM HAS A PORT

Most programs are registered against a port, a unique port per program instance, regardless of whether they use this port to communicate on or not (most don't). As a result, checking for missing programs is easily automated. Each program has a unique 'tag': its port allocation. In the mdspports database table, the running program, its start time, end time and its pid is stored. Periodically, every 5 minutes or so, each pid in the ports table is looked for, and if it is not found but an end time hasn't been entered, then the assumption is that it has crashed.

40.2 GLOBAL AND EXTERNAL VARIABLES

<System_Name> uses global variables very heavily. Do not *expect* any variables to be passed. The only time variables will be passed is when a function is in a module and will be called by a number of programs. If you are new to <System_Name> you had better get used to this.

40.3 RETURN TYPES

Approximately 0% of <System_Name> programs are declared with a return type, the return type being defaulted to int if the coder wants it to return something, including TRUE and FALSE (still typedef or #define'd ints in C).

40.4 SIGNATURES

Combining the previous two points one will see that most <System_Name> functions have no signature. As the default in C is that any referenced as yet un-declared function is taken to return an int and take no parameters, no forward references or C++ prototype header files need to be – or are – used. All (most) functions match their default prototypes.

40.5 MAKEFILES

40.5.1 LOCATION

The makefiles are in the .../c/makes/ directory, one for each program. The makefile for skymds06 is makefile.06 and that for the email gateway is makefile.emgw. Each makefile is called from the actual Makefile in the ~/c/ directory.

40.5.2 HEADER FILE INCLUSION

Also, makefiles do not contain an exhaustive list of header files as dependencies. Before you tear your hair out, remember that in many, even most, situations this will be OK. Any reference to a new variable in the source file will force a recompile, while changes to many (especially unused) aspects of included headers will be independent of program compilation or execution. And everything has to pass the linker.

Problems will arise however in areas such as:

- sizeof() calculations and operations
- activities that use the result of a sizeof()
- also note that the C-ISAM database tables a fundamentally free form data, templated by a header file. A header may change and while variable name access will be fine, operations based on record sizeof(), such as queue updates, can cause queue and / or database table corruptions.
- type changes of a variable such that type promotion fails, doesn't work, or is independent of the new type
- structure changes, where only the structure name is referred to in the code
- unions – where someone has tried to be clever in the code.
- index changes in database tables (described in the headers)

Personally, I would, in addition, include the makefile (itself) in the makefile's dependencies. You don't want to not re-add that new (old) library to your system.

Header files listed in skymds24l.c's makefile:

```
gentypes.h gettime.h macros.h mdsfldsz.h mdsparam.h mdsporec.h mdstrrec.h globvar.h mdshirec.h  
frc/frcsprec.h
```

Header files used in skymds24l.c's code:

```
gettime.h macros.h gentypes.h mdsfldsz.h mdsparam.h mdspol.h mdstrem.h security.h globvar.h  
startel/stafldsz.h frc/frcfldsz.h sel/selfldsz.h saf/saffldsz.h mdsgl3000.h mdssqrec.h mdscdr.h mdscdrec.h  
mdsporec.h mdstrrec.h mdsadrec.h mdsautrec.h mdsbtrec.h mdsckrec.h mdsc1rec.h mdscorec.h mdsanrec.h  
mdscsrec.h mdscstrec.h mdsurec.h mdsderek.h mdserrec.h mdsdsrec.h mdSDLrec.h mdsecrec.h mdsenrec.h  
mdsesrec.h mdsetrec.h mdsfarec.h mdsfmrec.h mdsttrec.h mdsfurec.h mdsfxrec.h mdshirec.h mdshsrec.h  
mdsiprec.h mdslirec.h mdsnrec.h mdsnrrec.h mdsparec.h mdspfrec.h mdpsrec.h mdspurec.h mdsnmrec.h  
mdsrprec.h mdsscrec.h mdsserec.h mdssfrec.h mdssqrec.h mdsssrec.h mdsstrec.h mdsttrec.h mdsttrec.h  
mdsttrec.h mdsunrec.h mdscprec.h mdssprec.h mdssdrec.h mdsglrec.h mdsg2rec.h mdsalrec.h mdsemblk.h  
mdsaprec.h frc/frcfimrec.h frc/frcetrec.h frc/frcenrec.h frc/frcsirec.h frc/frcsrec.h frc/frctrrec.h  
frc/frcmrec.h frc/frcsprec.h frc/frctsrec.h frc/frcelrec.h frc/frcmlrec.h frc/frcurec.h frc/frog1rec.h  
frc/frcscrec.h frc/frcogrec.h startel/staalrec.h startel/staa2rec.h sel/selserc.h sel/selcorec.h  
sel/selbrrec.h sel/selprrec.h sel/seladrec.h vip/vip1rec.h vip/vipa2rec.h saf/safprrec.h saf/safsprec.h  
saf/saflgrec.h
```

40.6 HARD CODED VARIABLES, VARIABLE NAMES

There are a lot of hard coded magic numbers and strings and, consequently, few constants.h type header files or constants.ini data files.

40.7 PROGRAM INDEPENDENCE

To quite a remarkable, and admirable, extent the 250 programs of the <System_Name> suite are pretty much independent of each other. Most either read to or write from a queue of some type, and they will generally continue to read messages (if there are any) and/or write them regardless of what most other programs are doing. True, some programs rely on, call, are called by others, but for such a large system – at the program level at least – the units (programs) are safe and independent from each other.

40.8 DATABASE TABLES

40.8.1.1 Data structures

<System_Name> uses an infoirmix C-ISAM database, an online manual for which is at

G:\Technology\Product Systems Development\<System_Name>1\DOCUMENT\C-ISAM Manual\C-isam_MAN.pdf

Each database table is practically a free form data set, the interpretation of which is controlled by mds*rec.h header files, e.g. mdsporec.h, for the ports database table header.

To read and write to / from these tables one:

- use a C-ISAM function to open the database table and return an integer file ID – like a normal operating system file
- clean out an instance of the appropriate data structure for this table.
- fill in the fields that one wants to search on
- use the appropriate C-ISAM function to return either the, or the first, record that contains the combination of data you filled in the previous step
- use the data from the record(s), or alter it and write it back to the table

40.8.1.2 Management

All the database tables used in a program are opened (open file pointers are constructed for them) in the initialisation function of each program. In a couple of heavily used programs (e.g. skymds01) a function / device is used to periodically close file pointers that haven't been used for a while.

40.9 GROUPING, HANDLING MAINTENANCE FUNCTIONS

There is a collection of core <System_Name> contact maintenance functions that share the following properties

- They are prefixed by a three letter contact code, e.g. REM_add_reminders()
- All these core <System_Name> maintenance functions are grouped together in code files named mds*code.c*, e.g. mdsrem.c for all the core reminder maintenance functions
- These functions are called mostly from program specific intermediate handling files. Following the reminder example, skymds27 uses a whole range of reminder functions, all coded in mds27*code.c*, e.g. mds27rem.c. These functions are not prefixed, having more descriptive names (generate_reminder() for example).
- At the top level, each daemon has its own function to prepare and call the intermediate functions in the mds??*code.c* file. This could be process_reminders()

40.9.1 A CALL GRAPH EXAMPLE

```
process_reminders( )...                                in, say skymds27
    calls generateReminder( ),..., which               in mds27rem.h
        calls REM_add_reminders( )                   in mdsrem.c
```

40.9.2 THE LEVELS

At the top level, the programs skymds01, skymds06, and skymds27 currently implement this scheme.

At the second level, functions in the following source code files are used:

mds01adv.c	mds06saf.c	mds0jadv.c	mds27adv.c
mds01ans.c		mds0jcom.c	mds27cpa.c
mds01con.c		mds0jcpa.c	mds27eml.c
mds01cpa.c		mds0jeml.c	mds27esc.c
mds01edb.c		mds0jesc.c	mds27fax.c
mds01evt.c		mds0jfax.c	mds27frm.c
mds01frm.c		mds0jfrm.c	mds27grp.c
mds01ini.c		mds0jgrp.c	mds27pag.c
mds01man.c		mds0jrem.c	mds27phn.c
mds01rem.c		mds0jros.c	mds27rem.c
mds01sch.c		mds0jsta.c	mds27ros.c
mds01srn.c		mds0jtms.c	mds27tms.c
		mds0jtxt.c	mds27txt.c

At the third, lowest level, these source code files have the contact utility functions

```
mdsadv.c
mdscat.c
mdsdr.c
mdsepa.c
mdsedb.c
mdsfrm.c
mdslog.c
mdsmnt.c
mdsmtp.c
mdsrem.c
mdssch.c
mdsshm.c
```

40.9.3 AN EXTENSION OF THIS NAMING PATTERN

A few other function groups have similar names: e.g. PORT_set_port_started(). Maybe, if <System_Name> becomes Object Orientated each of these will become part of a XXX objects (e.g. **port** or **reminder** (for REM)).

40.10 INITIALISE FUNCTIONS

Most (98%) of the <System_Name> applications have initialise functions in a format very similar to the following

- read in the required environment variables.
- parse the command-line.
- use background() to put the process into the background.
- catch a few – or many – signals.
- use PORT_set_port_started() to record in the mdspports table that skynds23 has started – and what port it is registered against.
- obtain and test the shutdown semaphore.
- get the sleep time for the function
- open all the database tables an other files required.
- attach to the shared memory blocks required
- write a log message that this program has started.

<System_Name> Code file example: skymds06

Following is an example of a typical <System_Name> source-code file.

skymds06 is used to process events from the timed events database table and write appropriate messages to the <System_Name> bulk queue.

[twenty pages of source code have been removed]