```
**********************************************************
   98035 Wed May 20 11:44:21 2015
new/usr/src/cmd/make/bin/main.cc
make: unifdef for _CHECK_UPDATE_H (undefined)
**********************************************************
______unchanged_portion_omitted_
#endif

extern  Name            normalize_name(register wchar_t *name_string, register i

extern  int             main(int, char * []);

static  void            append_makeflags_string(Name, String);
static  void            doalarm(int);
static  void            enter_argv_values(int , char **, ASCII_Dyn_Array *);
static  void            make_targets(int, char **, Boolean);
static  int             parse_command_option(char);
static  void            read_command_options(int, char **);
static  void            read_environment(Boolean);
static  void            read_files_and_state(int, char **);
static  Boolean         read_makefile(Name, Boolean, Boolean, Boolean);
static  void            report_recursion(Name);
static  void            set_sgs_support(void);
static  void            setup_for_projectdir(void);
static  void            setup_makeflags_argv(void);
static  void            report_dir_enter_leave(Boolean entering);

extern void expand_value(Name, register String , Boolean);

#ifdef DISTRIBUTED
        extern  int             dmake_ofd;
        extern  FILE*           dmake_ofp;
        extern  int             rxmPid;
        extern  XDR             xdrs_out;
#endif
#ifdef TEAMWARE_MAKE_CMN
        extern  char            verstring[];
#endif

jmp_buf jmpbuffer;
extern nl_catd catd;

/*
 *      main(argc, argv)
 *
 *      Parameters:
 *              argc                    You know what this is
 *              argv                    You know what this is
 *
 *      Static variables used:
 *              list_all_targets        make -T seen
 *              trace_status            make -p seen
 *
 *      Global variables used:
 *              debug_level             Should we trace make actions?
 *              keep_state              Set if .KEEP_STATE seen
 *              makeflags               The Name "MAKEFLAGS", used to get macro
 *              remote_command_name     Name of remote invocation cmd ("on")
 *              running_list            List of parallel running processes
 *              stdout_stderr_same      true if stdout and stderr are the same
 *              auto_dependencies       The Name "SUNPRO_DEPENDENCIES"
 *              temp_file_directory     Set to the dir where we create tmp file
 *              trace_reader            Set to reflect tracing status
 *              working_on_targets      Set when building user targets
 */
int
```

```
main(int argc, char *argv[])
{
        /*
         * cp is a -> to the value of the MAKEFLAGS env var,
         * which has to be regular chars.
         */
        register char           *cp;
        char                    make_state_dir[MAXPATHLEN];
        Boolean                 parallel_flag = false;
        char                    *prognameptr;
        char                    *slash_ptr;
        mode_t                  um;
        int                     i;
#ifdef TEAMWARE_MAKE_CMN
        struct itimerval        value;
        char                    def_dmakerc_path[MAXPATHLEN];
        Name                    dmake_name, dmake_name2;
        Name                    dmake_value, dmake_value2;
        Property                prop, prop2;
        struct stat             statbuf;
        int                     statval;
#endif

        struct stat             out_stat, err_stat;
        hostid = gethostid();
#ifdef TEAMWARE_MAKE_CMN
        avo_get_user(NULL, NULL); // Initialize user name
#endif
        bsd_signals();

        (void) setlocale(LC_ALL, "");


#ifdef DMAKE_STATISTICS
        if (getenv(NOCATGETS("DMAKE_STATISTICS"))) {
                getname_stat = true;
        }
#endif


        /*
         * avo_init() sets the umask to 0.  Save it here and restore
         * it after the avo_init() call.
         */
#if defined(TEAMWARE_MAKE_CMN) || defined(MAKETOOL)
        um = umask(0);
        avo_init(argv[0]);
        umask(um);

        cleanup = new Avo_cleanup(NOCATGETS("dmake"), argc, argv);
#endif

#if defined(TEAMWARE_MAKE_CMN)
        catd = catopen(AVO_DOMAIN_DMAKE, NL_CAT_LOCALE);
        libcli_init();

#ifdef _CHECK_UPDATE_H
        /* This is for dmake only (not for Solaris make).
         * Check (in background) if there is an update (dmake patch)
         * and inform user
         */
        {
                Avo_err                 *err;
                char                    *dir;
                err = avo_find_run_dir(&dir);
                if (AVO_OK == err) {
                        AU_check_update_service(NOCATGETS("Dmake"), dir);
                }
        }
#endif /* _CHECK_UPDATE_H */
#endif

// ---> fprintf(stderr, catgets(catd, 15, 666, "--- SUN make ---\n"));


#if defined(TEAMWARE_MAKE_CMN) || defined(MAKETOOL)
/*
 * I put libmksdmsi18n_init() under #ifdef because it requires avo_i18n_init()
 * from avo_util library.
 */
        libmksdmsi18n_init();
#endif


#ifndef TEAMWARE_MAKE_CMN
        textdomain(NOCATGETS("SUNW_SPRO_MAKE"));
#endif /* TEAMWARE_MAKE_CMN */

#ifdef TEAMWARE_MAKE_CMN
        g_argc = argc;
        g_argv = (char **) malloc((g_argc + 1) * sizeof(char *));
        for (i = 0; i < argc; i++) {
                g_argv[i] = argv[i];
        }
        g_argv[i] = NULL;
#endif /* TEAMWARE_MAKE_CMN */

        /*
         * Set argv_zero_string to some form of argv[0] for
         * recursive MAKE builds.
         */

        if (*argv[0] == (int) slash_char) {
                /* argv[0] starts with a slash */
                argv_zero_string = strdup(argv[0]);
        } else if (strchr(argv[0], (int) slash_char) == NULL) {
                /* argv[0] contains no slashes */
                argv_zero_string = strdup(argv[0]);
        } else {
                /*
                 * argv[0] contains at least one slash,
                 * but doesn't start with a slash
                 */
                char    *tmp_current_path;
                char    *tmp_string;

                tmp_current_path = get_current_path();
                tmp_string = getmem(strlen(tmp_current_path) + 1 +
                                    strlen(argv[0]) + 1);
                (void) sprintf(tmp_string,
                               "%s/%s",
                               tmp_current_path,
                               argv[0]);
                argv_zero_string = strdup(tmp_string);
                retmem_mb(tmp_string);
        }

        /*
         * The following flags are reset if we don't have the
         * (.nse_depinfo or .make.state) files locked and only set
         * AFTER the file has been locked. This ensures that if the user
         * interrupts the program while file_lock() is waiting to lock
```

```
         * the file, the interrupt handler doesn't remove a lock
         * that doesn't belong to us.
         */
        make_state_lockfile = NULL;
        make_state_locked = false;


        /*
         * look for last slash char in the path to look at the binary
         * name. This is to resolve the hard link and invoke make
         * in svr4 mode.
         */

        /* Sun OS make standart */
        svr4 = false;
        posix = false;
        if(!strcmp(argv_zero_string, NOCATGETS("/usr/xpg4/bin/make"))) {
                svr4 = false;
                posix = true;
        } else {
                prognameptr = strrchr(argv[0], '/');
                if(prognameptr) {
                        prognameptr++;
                } else {
                        prognameptr = argv[0];
                }
                if(!strcmp(prognameptr, NOCATGETS("svr4.make"))) {
                        svr4 = true;
                        posix = false;
                }
        }
        if (getenv(USE_SVR4_MAKE) || getenv(NOCATGETS("USE_SVID"))){
           svr4 = true;
           posix = false;
        }

        /*
         * Find the dmake_compat_mode: posix, sun, svr4, or gnu_style, .
         */
        char * dmake_compat_mode_var = getenv(NOCATGETS("SUN_MAKE_COMPAT_MODE"))
        if (dmake_compat_mode_var != NULL) {
                if (0 == strcasecmp(dmake_compat_mode_var, NOCATGETS("GNU"))) {
                        gnu_style = true;
                }
                //svr4 = false;
                //posix = false;
        }

        /*
         * Temporary directory set up.
         */
        char * tmpdir_var = getenv(NOCATGETS("TMPDIR"));
        if (tmpdir_var != NULL && *tmpdir_var == '/' && strlen(tmpdir_var) < MAX
                strcpy(mbs_buffer, tmpdir_var);
                for (tmpdir_var = mbs_buffer+strlen(mbs_buffer);
                        *(--tmpdir_var) == '/' && tmpdir_var > mbs_buffer;
                        *tmpdir_var = '\0');
                if (strlen(mbs_buffer) + 32 < MAXPATHLEN) { /* 32 = strlen("/dma
                        sprintf(mbs_buffer2, NOCATGETS("%s/dmake.tst.%d.XXXXXX")
                                mbs_buffer, getpid());
                        int fd = mkstemp(mbs_buffer2);
                        if (fd >= 0) {
                                close(fd);
                                unlink(mbs_buffer2);
                                tmpdir = strdup(mbs_buffer);
                        }
                }
        }

        /* find out if stdout and stderr point to the same place */
        if (fstat(1, &out_stat) < 0) {
                fatal(catgets(catd, 1, 165, "fstat of standard out failed: %s"),
        }
        if (fstat(2, &err_stat) < 0) {
                fatal(catgets(catd, 1, 166, "fstat of standard error failed: %s"
        }
        if ((out_stat.st_dev == err_stat.st_dev) &&
            (out_stat.st_ino == err_stat.st_ino)) {
                stdout_stderr_same = true;
        } else {
                stdout_stderr_same = false;
        }
        /* Make the vroot package scan the path using shell semantics */
        set_path_style(0);

        setup_char_semantics();

        setup_for_projectdir();

        /*
         * If running with .KEEP_STATE, curdir will be set with
         * the connected directory.
         */
        (void) atexit(cleanup_after_exit);

        load_cached_names();

/*
 *      Set command line flags
 */
        setup_makeflags_argv();
        read_command_options(mf_argc, mf_argv);
        read_command_options(argc, argv);
        if (debug_level > 0) {
                cp = getenv(makeflags->string_mb);
                (void) printf(catgets(catd, 1, 167, "MAKEFLAGS value: %s\n"), cp
        }

        setup_interrupt(handle_interrupt);

        read_files_and_state(argc, argv);

#ifdef TEAMWARE_MAKE_CMN
        /*
         * Find the dmake_output_mode: TXT1, TXT2 or HTML1.
         */
        MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_OUTPUT_MODE"));
        dmake_name2 = GETNAME(wcs_buffer, FIND_LENGTH);
        prop2 = get_prop(dmake_name2->prop, macro_prop);
        if (prop2 == NULL) {
                /* DMAKE_OUTPUT_MODE not defined, default to TXT1 mode */
                output_mode = txt1_mode;
        } else {
                dmake_value2 = prop2->body.macro.value;
                if ((dmake_value2 == NULL) ||
                    (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("TXT1")))) {
                        output_mode = txt1_mode;
                } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("TXT2")))
                        output_mode = txt2_mode;
                } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("HTML1"))
                        output_mode = html1_mode;
                } else {
```

```
                        warning(catgets(catd, 1, 352, "Unsupported value '%s' fo
                              dmake_value2->string_mb);
                }
        }
        /*
         * Find the dmake_mode: distributed, parallel, or serial.
         */
    if ((!pmake_cap_r_specified) &&
        (!pmake_machinesfile_specified)) {
        MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_MODE"));
        dmake_name2 = GETNAME(wcs_buffer, FIND_LENGTH);
        prop2 = get_prop(dmake_name2->prop, macro_prop);
        if (prop2 == NULL) {
                /* DMAKE_MODE not defined, default to distributed mode */
                dmake_mode_type = distributed_mode;
                no_parallel = false;
        } else {
                dmake_value2 = prop2->body.macro.value;
                if ((dmake_value2 == NULL) ||
                    (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("distributed"))
                        dmake_mode_type = distributed_mode;
                        no_parallel = false;
                } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("parallel
                        dmake_mode_type = parallel_mode;
                        no_parallel = false;
                } else if (IS_EQUAL(dmake_value2->string_mb, NOCATGETS("serial")
                        dmake_mode_type = serial_mode;
                        no_parallel = true;
                } else {
                        fatal(catgets(catd, 1, 307, "Unknown dmake mode argument
                }
        }

        if ((!list_all_targets) &&
            (report_dependencies_level == 0)) {
                /*
                 * Check to see if either DMAKE_RCFILE or DMAKE_MODE is defined.
                 * They could be defined in the env, in the makefile, or on the
                 * command line.
                 * If neither is defined, and $(HOME)/.dmakerc does not exists,
                 * then print a message, and default to parallel mode.
                 */
#ifdef DISTRIBUTED
                MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_RCFILE"));
                dmake_name = GETNAME(wcs_buffer, FIND_LENGTH);
                MBSTOWCS(wcs_buffer, NOCATGETS("DMAKE_MODE"));
                dmake_name2 = GETNAME(wcs_buffer, FIND_LENGTH);
                if ((((prop = get_prop(dmake_name->prop, macro_prop)) == NULL) |
                     ((dmake_value = prop->body.macro.value) == NULL)) &&
                    (((prop2 = get_prop(dmake_name2->prop, macro_prop)) == NULL)
                     ((dmake_value2 = prop2->body.macro.value) == NULL))) {
                        Boolean empty_dmakerc = true;
                        char *homedir = getenv(NOCATGETS("HOME"));
                        if ((homedir != NULL) && (strlen(homedir) < (sizeof(def_
                                sprintf(def_dmakerc_path, NOCATGETS("%s/.dmakerc
                                if ((((statval = stat(def_dmakerc_path, &statbuf
                                        ((statval == 0) && (statbuf.st_size == 0
                                } else {
                                        Avo_dmakerc     *rcfile = new Avo_dmaker
                                        Avo_err         *err = rcfile->read(def_
                                        if (err) {
                                                fatal(err->str);
                                        }
                                        empty_dmakerc = rcfile->was_empty();
                                        delete rcfile;
                                }
                        }
                        if (empty_dmakerc) {
                                if (getenv(NOCATGETS("DMAKE_DEF_PRINTED")) == NU
                                        putenv(NOCATGETS("DMAKE_DEF_PRINTED=TRUE
                                        (void) fprintf(stdout, catgets(catd, 1,
                                        (void) fprintf(stdout, catgets(catd, 1,
                                }
                                dmake_mode_type = parallel_mode;
                                no_parallel = false;
                        }
                }
#else
                if(dmake_mode_type == distributed_mode) {
                        (void) fprintf(stdout, NOCATGETS("dmake: Distributed mod
                        (void) fprintf(stdout, NOCATGETS("       Defaulting to p
                        dmake_mode_type = parallel_mode;
                        no_parallel = false;
                }
#endif  /* DISTRIBUTED */
        }
    }
#endif

#ifdef TEAMWARE_MAKE_CMN
        parallel_flag = true;
        /* XXX - This is a major hack for DMake/Licensing. */
        if (getenv(NOCATGETS("DMAKE_CHILD")) == NULL) {
                if (!avo_cli_search_license(argv[0], dmake_exit_callback, TRUE,
                        /*
                         * If the user can not get a TeamWare license,
                         * default to serial mode.
                         */
                        dmake_mode_type = serial_mode;
                        no_parallel = true;
                } else {
                        putenv(NOCATGETS("DMAKE_CHILD=TRUE"));
                }
                start_time = time(NULL);
                /*
                 * XXX - Hack to disable SIGALRM's from licensing library's
                 *       setitimer().
                 */
                value.it_interval.tv_sec = 0;
                value.it_interval.tv_usec = 0;
                value.it_value.tv_sec = 0;
                value.it_value.tv_usec = 0;
                (void) setitimer(ITIMER_REAL, &value, NULL);
        }

//
// If dmake is running with -t option, set dmake_mode_type to serial.
// This is done because doname() calls touch_command() that runs serially.
// If we do not do that, maketool will have problems.
//
        if(touch) {
                dmake_mode_type = serial_mode;
                no_parallel = true;
        }
#else
        parallel_flag = false;
#endif

#if defined (TEAMWARE_MAKE_CMN) && defined(REDIRECT_ERR)
        /*
         * Check whether stdout and stderr are physically same.
         * This is in order to decide whether we need to redirect
```

```
         * stderr separately from stdout.
         * This check is performed only if __DMAKE_SEPARATE_STDERR
         * is not set. This variable may be used in order to preserve
         * the 'old' behaviour.
         */
        out_err_same = true;
        char * dmake_sep_var = getenv(NOCATGETS("__DMAKE_SEPARATE_STDERR"));
        if (dmake_sep_var == NULL || (0 != strcasecmp(dmake_sep_var, NOCATGETS("
                struct stat stdout_stat;
                struct stat stderr_stat;
                if( (fstat(1, &stdout_stat) == 0)
                 && (fstat(2, &stderr_stat) == 0) )
                {
                        if( (stdout_stat.st_dev != stderr_stat.st_dev)
                         || (stdout_stat.st_ino != stderr_stat.st_ino) )
                        {
                                out_err_same = false;
                        }
                }
        }
#endif

#ifdef DISTRIBUTED
        /*
         * At this point, DMake should startup an rxm with any and all
         * DMake command line options. Rxm will, among other things,
         * read the rc file.
         */
        if ((!list_all_targets) &&
            (report_dependencies_level == 0) &&
            (dmake_mode_type == distributed_mode)) {
                startup_rxm();
        }
#endif

/*
 *      Enable interrupt handler for alarms
 */
        (void) bsd_signal(SIGALRM, (SIG_PF)doalarm);

/*
 *      Check if make should report
 */
        if (getenv(sunpro_dependencies->string_mb) != NULL) {
                FILE    *report_file;

                report_dependency("");
                report_file = get_report_file();
                if ((report_file != NULL) && (report_file != (FILE*)-1)) {
                        (void) fprintf(report_file, "\n");
                }
        }

/*
 *      Make sure SUNPRO_DEPENDENCIES is exported (or not) properly.
 */
        if (keep_state) {
                maybe_append_prop(sunpro_dependencies, macro_prop)->
                  body.macro.exported = true;
        } else {
                maybe_append_prop(sunpro_dependencies, macro_prop)->
                  body.macro.exported = false;
        }

        working_on_targets = true;
        if (trace_status) {
                dump_make_state();
                fclose(stdout);
                fclose(stderr);
                exit_status = 0;
                exit(0);
        }
        if (list_all_targets) {
                dump_target_list();
                fclose(stdout);
                fclose(stderr);
                exit_status = 0;
                exit(0);
        }
        trace_reader = false;

        /*
         * Set temp_file_directory to the directory the .make.state
         * file is written to.
         */
        if ((slash_ptr = strrchr(make_state->string_mb, (int) slash_char)) == NU
                temp_file_directory = strdup(get_current_path());
        } else {
                *slash_ptr = (int) nul_char;
                (void) strcpy(make_state_dir, make_state->string_mb);
                *slash_ptr = (int) slash_char;
                   /* when there is only one slash and it's the first
                   ** character, make_state_dir should point to '/'.
                   */
                if(make_state_dir[0] == '\0') {
                   make_state_dir[0] = '/';
                   make_state_dir[1] = '\0';
                }
                if (make_state_dir[0] == (int) slash_char) {
                        temp_file_directory = strdup(make_state_dir);
                } else {
                        char    tmp_current_path2[MAXPATHLEN];

                        (void) sprintf(tmp_current_path2,
                                       "%s/%s",
                                       get_current_path(),
                                       make_state_dir);
                        temp_file_directory = strdup(tmp_current_path2);
                }
        }

#ifdef DISTRIBUTED
        building_serial = false;
#endif

        report_dir_enter_leave(true);

        make_targets(argc, argv, parallel_flag);

        report_dir_enter_leave(false);

        if (build_failed_ever_seen) {
                if (posix) {
                        exit_status = 1;
                }
                exit(1);
        }
        exit_status = 0;
        exit(0);
        /* NOTREACHED */
}
```
______***unchanged_portion_omitted_***

```
**********************************************************
   53472 Wed May 20 11:44:22 2015
new/usr/src/cmd/make/bin/parallel.cc
make: unifdef for _CHECK_UPDATE_H (undefined)
**********************************************************
______unchanged_portion_omitted_

/*
 *      await_parallel(waitflg)
 *
 *      Waits for parallel children to exit and finishes their processing.
 *      If waitflg is false, the function returns after update_delay.
 *
 *      Parameters:
 *              waitflg         dwight
 */
void
await_parallel(Boolean waitflg)
{
#ifdef _CHECK_UPDATE_H
        static int number_of_unknown_children = 0;
#endif /* _CHECK_UPDATE_H */
        Boolean         nohang;
        pid_t           pid;
        int             status;
        Running         rp;
        int             waiterr;

        nohang = false;
        for ( ; ; ) {
                if (!nohang) {
                        (void) alarm((int) update_delay);
                }
                pid = waitpid((pid_t)-1,
                              &status,
                              nohang ? WNOHANG : 0);
                waiterr = errno;
                if (!nohang) {
                        (void) alarm(0);
                }
                if (pid <= 0) {
                        if (waiterr == EINTR) {
                                if (waitflg) {
                                        continue;
                                } else {
                                        return;
                                }
                        } else {
                                return;
                        }
                }
                for (rp = running_list;
                     (rp != NULL) && (rp->pid != pid);
                     rp = rp->next) {
                        ;
                }
                if (rp == NULL) {
#ifdef _CHECK_UPDATE_H
                        /* Ignore first child - it is check_update */
                        if (number_of_unknown_children <= 0) {
                                number_of_unknown_children = 1;
                                return;
                        }
#endif /* _CHECK_UPDATE_H */
                        if (send_mtool_msgs) {
                                continue;
```

```
                        } else {
                                fatal(catgets(catd, 1, 128, "Internal error: ret
                        }
                } else {
                        rp->state = (WIFEXITED(status) && WEXITSTATUS(status) ==
                }
                nohang = true;
                parallel_process_cnt--;

#if defined (TEAMWARE_MAKE_CMN) && defined (MAXJOBS_ADJUST_RFE4694000)
                if (job_adjust_mode == ADJUST_M2) {
                        if (m2_release_job()) {
                                job_adjust_error();
                        }
                }
#endif
        }
}
______unchanged_portion_omitted_
```